(12) UK Patent Application (19) GB (11) 2 427 722 (13) A

(43) Date of A Publication 03.01.2007

(21) Application No: 0524772.1

(22) Date of Filing: 05.12.2005

(30) Priority Data:
(31) 0509738 (32) 12.05.2005 (33) GB

(71) Applicant(s):
**Cambridge Consultants Limited**
**(Incorporated in the United Kingdom)**
**Science Park, Milton Road, CAMBRIDGE,**
**Cambs, CB4 0DW, United Kingdom**

(72) Inventor(s):
**Alistair Guy Morfey**
**Karl Leighton Swepson**
**Martin David Cooper**
**Alan Mycroft**
**Neil Edward Johnson**

(74) Agent and/or Address for Service:
**Mathys & Squire**
**120 Holborn, LONDON, EC1N 2SQ,**
**United Kingdom**

(51) INT CL:
***G06F 15/173*** (2006.01) ***G06F 1/24*** (2006.01)
***G06F 11/10*** (2006.01)

(52) UK CL (Edition X ):
**G4A** AMP APCH

(56) Documents Cited:
**GB 2329984 A** **GB 2217059 A**
**GB 2156550 A** **US 5870602 B**
**US 5815144 B**

(58) Field of Search:
UK CL (Edition X ) **G4A**
INT CL **G06F**
Other: **EPODOC, WPI.**

(54) Abstract Title: **Computer system, processor and interface to other processors.**

(57) Disclosed is a computer system with a master computer, control pods, acting as interface units, and integrated circuits each with at least one processor. The master computer is able to interact with any of the processors via the control pods. The integrated circuits may provide an interface for the control pods to signal a request for access to the storage locations of an associated processor and to provide parity checking for the accesses. The master computer may be able to interact with any of the processors to reset a portion of the integrated circuit. The master computer may be able to issue cancellation instructions to any of the processors to cancel previous instructions. The system, may, when requested data is sent by a processor output a time stamp indicating when the data was read or written. The master computer may query the integrated circuits to identify the register sizes and processor numbers. The control pods may spilt data received form the master computer in to fragments or packages and collate date packages received form elsewhere before sending on to the master computer. Also disclosed are data processing apparatus with a plurality of processors, apparatus with a processor with a controller that generates odd and even parity over an even number of bits, apparatus with a processor with a control circuit to reset the apparatus, apparatus with a processor with a counter, apparatus with a processor with a control circuit to cancel pending requests, apparatus with a processor with a time stamp, apparatus with a processor with analogue acquisition circuitry, and a processor able to read data from adjacent memory register pairs.

GB 2 427 722 A

The claims were filed later than the filing date but within the period prescribed by Rule 25(1) of the Patents Rules 1995.

**Original Printed on Recycled Paper**

2427722

**Processor and Interface**

The present invention relates to a processor, to an interface associated with the processor, and to methods of operation of the processor and the interface. The invention also relates to an instruction set and to methods of programming operation of the processors and the interface. The invention also relates to a data processing apparatus. The invention also relates to a family of processors.

Broadly, according to an aspect of the invention, there is provided a family of processors, which share a number of common characteristics, and in which substantially the only differences between the processors are as a result of the different sizes thereof.

According to another aspect of the invention, there is provided a processor forming part of a family of similar processors each having a number of common characteristics, and in which the only differences between the processors and the other processors in the family are as a result of the different sizes of the processors.

Preferably, the processors are similar in that that they share one or more of the characteristics listed in Table 1 below. More preferably, each of the processors is unique in accordance with one or more of the parameters listed in Table 1 below.

Broadly, according to another aspect of the invention, there is provided a data processing apparatus including one or more processors, each processor having associated memory, registers and debug instructions, and an interface capable of communicating with the or each processor, wherein the interface provides non-invasive access to the or each processor memory, registers or debug instructions.

Preferably, the data processing apparatus is implemented in a single semiconductor device or chip, which contains one interface and one or more processors. The processors are preferably slaves to the interface. More

preferably, the processors form part of the family of processors as hereinbefore described.

The term "semiconductor device" as used herein is preferably intended to connote an ASIC or FPGA implemented in silicon or germanium or gallium-arsenide or some other semiconductor material. The semiconductor device is also referred to as a chip.

Preferably, the processors are optimised for embedded use in ASICs and FPGAs. More preferably, the processors may be of different sizes.

Preferably, the interface is optimised for embedded use in ASICs and FPGAs.

Preferably, the interface, which is also referred to as a SIF interface, can support an off-chip master and an on-chip master. Preferably, the off-chip master uses an external (xSIF) interface. Preferably, the on-chip master uses an internal (iSIF) interface. In this way, the interface may be controlled, and hence provide access to the processor's memory, registers and debug instructions provided, either via an external source (using an off-chip master), or via an on-chip device, such as a further processor. Further details in this regard can be found in the diagrams in Sections 2 and 11 of the SIF Definition document.

Preferably, the masters can read and write the registers and memory of one or more on-chip processors. More preferably, the masters can also issue debug instructions to one or more on-chip processors. Preferably, the external interface is a serial interface. Preferably, the internal interface is a parallel interface.

Preferably, the interface comprises arbitration hardware for ensuring that communication bandwidth is shared between the internal and external masters.

Preferably, the processors are those described in Table 1.

| | *Processor 1 (XAP3)* | *Processor 2 (XAP4)* | *Processor 3 (XAP5)* |
|---|---|---|---|
| *Preferred Memory Width* | 32 bit | 16 bit | 16 bit |
| *Data Bus* | 32 bit | 16 bit | 16 bit |
| *Address Bus* | 32 bit | 16 bit | 24 bit |
| *Memory limit* | 4 GB | 64 kB | 16 MB |
| *Memory Organisation* | Little Endian | Little Endian | Little Endian |
| *Instructions* | 16 bit – 50%<br>32 bit – 50% | 16 bit – 87.5%<br>32 bit – 12.5% | 16 bit – 87.5%<br>32 bit – 12.5% |
| *xSIF Shift Register* | 88 bit | 52 bit | 68 bit |
| *iSIF Registers Address, Control, Data, Status* | 32, 8, 32, 16 bit | 16, 8, 16, 12 bit | 24, 8, 16, 20 bit |
| *Processor Modes* | 4 | 4 | 4 |
| *User Registers* | 16 * 32 bit, R0-R15 | 8 * 16 bit, R0-R7 | 8 * 16 bit, R0-R7 |
| *Stack Pointer* | R15 | R7 | R7 |
| *Link Register* | R14 | R6 | Dedicated 24 bit LR |
| *Program Counter* | Dedicated 32 bit PC | Dedicated 16 bit PC | Dedicated 24 bit PC |
| *Breakpoint Registers* | 16 * 32 | 1 * 16 | 4 * 24 |
| *int variable* | 32 bit | 16 bit | 16 bit |
| *Data Pointers* | 32 bit | 16 bit | 16 bit |
| *Function Pointers* | 32 bit | 16 bit | 32 bit |
| *Software Toolkit* | xIDE for XAP3 | xIDE for XAP4 | xIDE for XAP5 |
| *C Compiler* | xap3-gcc, xap3ncc | xap4-gcc | xap5-gcc |
| *C Library* | xap3-lib | xap4-lib | xap5-lib |
| *Assembler* | binutils | binutils | binutils |
| *Linker* | binutils | binutils | binutils |
| *Code* | PC-relative | PC-relative | PC-relative |
| *Const* | PC-relative | PC-relative | PC-relative |
| *Global Variables* | GP-relative | PC-relative | PC-relative |

TABLE 1: Processor Features

Furthermore, each of the processors in the family also share the following common features:

- The processors may all be Load-Store RISC processors.
- The processors may all support unaligned byte addressing.
- The processors may all have a Von Neumann architecture (having a single memory and/or memory bus for code and data).

- The processors may all be implemented in a synthesisable Verilog RTL (Register Transfer Level) language.
- The processors may all be optimised for code density so that a given program written in C will compile to be as small as possible (giving lower cost and power consumption, while running at a faster speed).
- The processors may all have a mixture of 16 and 32 bit instructions. Preferably, these instructions can be freely mixed and are evaluated on an instruction by instruction basis. More preferably, a user does not have to specify whether to use a 16 or 32 bit instruction at the time of coding.
- The processors may all use the same style of instruction mnemonics.
- The processors may all have the same 4 processor modes: User, Supervisor, Interrupt, Non-Maskable Interrupt (NMI).
- The processors may all use a SIF interface for software debugging and data acquisition. The SIF interface supports an internal parallel iSIF interface and an external serial xSIF interface.
- The processors may all use the same developer environment for software development. In preferred embodiments, the developer environment comprises the xIDE Toolkits.
- The processors may all have GCC (GNU Compiler Collection) compilers.
- The compilers and instruction sets for use with the processors may all be developed together for compatibility and optimal code density.
- The processors may have good hardware support for embedded operating systems.

Preferably, all code for these processors will be written in C or another high-level language. Preferably, the code will normally use 8, 16 and 32 bit fixed-point data. The code will not normally use floating-point data. Some of this code will be custom developed for the application, but much will be off-the-shelf standard product software. Advantageously, the family of processors (XAP3, XAP4 and XAP5) all have good features for porting 3rd-party software.

As can be seen from Table 1, and as discussed above, the processors (XAP3, XAP4 and XAP5) all have many aspects in common. This 'family' approach makes it possible for programmers easily to migrate code from one processor to another. It also makes it easy for programmers acquainted with one processor to learn how to use another processor in the family. These advantages are achieved because of the consistent philosophy applied across all the processors in the family.

The programmer will typically choose a particular processor based upon :

- required software size
- speed requirements
- cost (size of processor and memory) requirements
- power-consumption requirements

The XAP4 processor is the smallest in size. The XAP3 processor is the largest, and the XAP5 processor is medium-sized.

A key feature of all the processors is good code density for compiled C code. In all cases this is achieved by instruction sets that are a mixture of 16 and 32 bit:

- Some instructions are only available as 16 bit
- Some instructions are only available as 32 bit
- Some instructions are available in 16 bit and in 32 bit form

Preferably, the instruction mnemonics do not indicate whether an instruction is 16 or 32 bit. Thus, the programmer and compiler do not need to be aware of whether instructions are 16 or 32 bit. Only the assembler has to choose between 16 or 32 bit encodings for a particular instruction. This is much more convenient for the programmer as he always has access to all of the processor and all of the instructions (some processors have modes which only allow the programmer to access a subset of the processor's hardware and instructions). The 16 and 32 bit instructions can be freely mixed and are

evaluated by the hardware at run-time. This may provide good code-density while retaining a friendly programming interface.

Particular aspects of this invention, and particular embodiments and further details, are described with reference to the following documents, which are arranged in the order indicated below.

- A document relating to a first aspect of the invention (XAP3), including:
  - A document entitled: ***First Aspect: Processor and Interface (XAP3)*** (that document being numbered from 8 to 119).
- A document relating to a second aspect of the invention (XAP4), including:
  - A document entitled: ***Second Aspect: Processor and Interface (XAP4)*** (that document being numbered from 120 to 243)
- An Appendix, including:
  - A document presenting an outline or summary of some of the inventions covered, entitled: ***Summary of Inventions*** (that document being labelled C7245-TM-004 v0.7 and having 5 pages, the pages being numbered from page 3 of 10 to page 7 of 10).

  - Further documents relating to the first aspect of the invention (XAP3), including:
  - A document entitled: ***XAP3 Specification*** (that document being labelled C7245-S-001 v3.0, and having 151 pages, the pages being numbered from page 1 of 151 to page 151 of 151).
  - A document entitled: ***XAP3 Programmer's Manual*** (that document being labelled C7245-UM-002 v1.0 and having 291 pages, the pages being numbered from 1 of 291 to 291 of 291).

  - Further documents relating to the second aspect of the invention (XAP4), including:
  - A document entitled: ***XAP4 Specification*** (that document being labelled C7432-S-001 v2.0, and having 131 pages, the pages being numbered from page 1 of 131 to page 131 of 131).

- A document entitled: ***XAP4 Programmer's Manual*** (that document being labelled C7432-UM-002 v1.3 and having 224 pages, the pages being numbered from 1 of 224 to 224 of 224).
- A document entitled ***XAP4a Hardware Reference Manual*** (that document being labelled C7432-UM-011 v1.4 and having 47 pages, the pages being numbered from 1 of 47 to 47 of 47).
- A document entitled ***XAP4a Instruction Set Quick Reference Card*** (that document being labelled C7432-UM-005 v1.11 and having 4 pages, the pages being numbered from 1 of 4 to 4 of 4).

- Documents relating to SIF (serial interface) aspects, including

- A document entitled: the ***SIF (Serial Interface) Definition*** (that document being labelled C4862-TM-005 v3.0, and having 74 pages, the pages being numbered from page 1 of 74 to page 74 of 74).
- A document entitled: ***xSIF API*** (that document being labelled C4862-TM-016 v0.10 and having 43 pages, the pages being numbered from page 1 of 43 to page 43 of 43).

The above-mentioned documents relate to processors which may be members of the family of processors described above. Preferably, these are in the form of the XAP3, XAP4 and XAP5 processors. Furthermore, the abovementioned documents also relate to a SIF interface system, for instance an internal (iSIF) or external (xSIF) interface system in the preferred embodiment, which is used together with the described processors.

It should also be noted that the above two aspects of the invention, as described in the documents entitled ***First Aspect: Processor and Interface (XAP3)*** and ***Second Aspect: Processor and Interface (XAP4)*** and in the documents to which they refer are closely related, since they cover a family of closely related processors and an associated interface. Therefore, any, some and/or all features in one aspect may be applied to any, some and/or all features in the other aspect, in any appropriate combination.

## First Aspect: Processor and interface

## *(XAP3)*

The present invention relates to a processor, to an interface – in particular a serial interface - associated with the processor, and to methods of operation of the processor and the interface. The invention also relates to an instruction set and to methods of programming operation of the processor and the interface.

Particular aspects of the invention, and particular embodiments, are described in sections 1 to 38.

More details of embodiments are then provided in the following further documents which are included herein:-

- document entitled Summary of Inventions (that document being labelled C7245-TM-004 v0.7 and having 5 pages, the pages being numbered from page 3 of 10 to page 7 of 10)
- document entitled XAP3 Specification (that document being labelled C7245-S-001 v3.0, and having 151 pages, the pages being numbered from page 1 of 151 to page 151 of 151)
- document entitled XAP3 Programmer's Manual (that document being labelled C7245-UM-002 v1.0 and having 291 pages, the pages being numbered from 1 of 291 to 291 of 291)
- document entitled the SIF (Serial Interface) Definition (that document being labelled C4862-TM-005 v3.0, and having 74 pages, the pages being numbered from page 1 of 74 to page 74 of 74)
- document entitled xSIF API (that document being labelled C4862-TM-016 v0.10 and having 43 pages, the pages being numbered from page 1 of 43 to page 43 of 43

The following text references diagrams and text in the above sections, particularly the XAP3 Specification and SIF Definition documents.

Features disclosed in any of sections 1 to 38 or in any of the further sections may be combined in any appropriate combination.

A brief overview of aspects of the invention, and preferred embodiments, is first provided.

Sections 21 to 38 relate in particular to a processor, for instance a XAP3 processor in the preferred embodiment.

Sections 1 to 20 relate in particular to an interface system, for instance a serial interface (SIF) system in the preferred embodiment. The SIF processor interface system is used in relation to a number of processors, including the XAP3.

In SIF communications, in the preferred embodiment, a SIF Computer transfers data to and from one or more SIF Slaves via one or more SIF Pods. The SIF Slave is normally an integrated circuit.

Aspects of the SIF and associated processor are described in WO 96/09583, WO 03/048978, GB 2382889, United Kingdom Patent Application Nos. 9419246.5 and 0129144.2, each in the name of Cambridge Consultants Limited and each of which is hereby incorporated by reference.

SIF technology includes elements of the SIF inside the integrated circuit and external SIF elements found in the SIF Computer and SIF Pods (see for instance the SIF Definition document, sections 2.1 and 2.2 for an overview of aspects of the preferred embodiment ).

The XAP3 and SIF Slave module are normally implemented together in the same integrated circuit (XAP3 Specification document, 9.1). The integrated circuit may also contain other circuit elements. The integrated circuit may

contain more than one processor, but normally contains just one SIF Slave module.

The XAP3 is a 32-bit processor that may be implemented in an integrated circuit. The integrated circuit may be a standard component or an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA).

In embodiments of the XAP3 processor, the same width is used for data, registers and address and instruction, in accordance with good processor design, and preferably the same memory can be used to store data and store programs and store pointers.

The XAP3 processor is described in detail in the XAP3 Specification document. The Programmer's model for the XAP3 is described in section 5 of the XAP3 Specification. The Instruction Set is described in section 8. The specific SIF Slave module used in the XAP3 is described in section 9 .

The preferred embodiment of a SIF system comprises three main components: a SIF Slave module (which may also be referred to herein simply as the SIF module or SIF, in various contexts), usually located in an integrated circuit; a SIF Pod, located external to the integrated circuit and preferably located external to the circuit board upon which the integrated circuit is located; and a SIF computer (also referred to as a computer or control computer) which includes a SIF driver, preferably implemented in software . The SIF Pod and the SIF computer together may be referred to as a SIF master. Alternatively the SIF Pod alone is sometimes referred to as the SIF master. The SIF computer may be a laptop or desktop computer and may be, for instance, a PC or Mac.

An overview of the architecture of preferred embodiments of the SIF processor interface system can be seen in the figure shown in section 2.2 (p8) of the SIF Definition document and in section 9.1 of the XAP3 Specification document.

The integrated circuit is referred to at various places herein as the SIF Slave ASIC (even though it may be an FPGA or standard component).

In the preferred embodiment, the SIF Slave ASIC contains a SIF Slave module, one or more processors, application-specific digital circuitry and application-specific analogue circuitry. In the preferred embodiment the processor is a XAP3 processor. In other embodiments the processor may be, for instance, a XAP1 or XAP2 processor.

A plurality of SIF Slave ASICs (or FPGAs) may be connected to a single SIF Pod, and a plurality of SIF Pods may be connected to a single SIF Computer, as can be seen for instance in the figure shown in section 2.1 of the SIF Definition document.

The SIF Slave module in the preferred embodiment comprises a shift register with three interface signals (SIF_CLK, SIF_MOSI, SIF_MISO) and handshake circuitry with one interface signal (SIF_LOADB). See section 7 of the SIF Definition document. The shift register has two inputs; SIF_CLK (to which a clock signal is applied to effect the movement of binary data between subsequent stages of the shift register) and SIF_MOSI (Master-Out, Slave-In) for receiving binary information in a serial manner from an external source (thereby allowing the information to be entered into the shift register and debug system). The shift register has a single output; SIF_MISO (Master-In, Slave-Out) for allowing binary data to be output in a serial manner from the shift register to an external source. In preferred examples, the shift register is a 64-bit shift register (XAP2) or an 88 bit shift register (XAP3). Typically, the SIF_CLK, SIF_MOSI, SIF_MISO and SIF_LOADB signals connect the SIF Slave module to the SIF Pod.

The SIF Pod may comprise a hardware-based interface between the SIF Slave ASIC and a personal computer. The SIF Pod may be connected to the SIF Slave ASIC by a 16-wire interface (see SIF Definition document, section

3) and may be connected to the PC via a standard communications interface such as Ethernet, RS232 or USB (Universal Serial Bus)

The SIF Driver in the SIF Computer may comprise a computer program that translates information between the standard communications interface (such as Ethernet, RS232 or USB) and the xSIF API (Application Programmer's Interface). This enables application programs on the PC to simply control and transfer information with the SIF Slave ASICs. Further computer programs may also be executed on the PC, which programs may utilise the xSIF API to allow data to be written to and read from the SIF Slave devices. This data may then be analysed to determine the operation of the processor, such as for allowing debugging of programs being executed by the processor or processors in the SIF Slave ASIC. Alternatively or additionally, data read from the SIF Slave ASIC may be data relating to the operation of a sensor or other device and the SIF Slave ASIC may then operate as a data logging device.

The SIF interface system can be operated on the SIF Slave ASIC concurrently with other software being executed by the ASIC processor, such as an operating system and user-mode programs. Concurrent operation may be achieved by time-slicing, which provides time windows during which the ASIC processor performs NOP (No Operation) leaving the SIF Slave module free to execute debug control of the processor, or to access the processor's memory and registers.

Typically each processor comprises a Core, an MMU (Memory Management Unit), an IVC (Interrupt Vector Controller) and a SIF Slave Module. See the XAP3 Specification document (section 4, p 21). The Core communicates with memory external to the processor via the Memory Management Unit. In a preferred example, the Memory Management Unit can communicate with external memory including Flash memory (typically used for the storage of programs), RAM (Random Access Memory, typically used for the temporary storage of data variables to be manipulated by programs) and one or more registers.

The memory management unit typically has address and read-write lines and data lines. These signals are typically uni-directional when on-chip. The data lines are typically bi-directional when off-chip.

The SIF Slave Module in the preferred embodiment is connected to the Memory Management Unit, such that the SIF Slave module is able to control the memory control and address signals, thereby enabling it to read and write data from/to the processor's memory.

In the preferred embodiment, a number of further lines are provided to enable interaction of the processor Core with the SIF Slave module.

In the preferred embodiment a number of lines are provided between the SIF Slave module and the SIF Pod, including the SIF_CLK, SIF_MOSI, SIF_MISO and SIF_LOADB lines referred to above. A further input to the SIF Slave ASIC is the chip select line SIF_CS. The 16-wire interface from the SIF Pod contains 4 lines that may be used as separate SIF_CS lines to up to 4 SIF Slave ASICs.

## 1. Asynchronous SIF handshake

The SIF Computer can still communicate with the SIF Slave ASIC at high data rates, even when the SIF Slave ASIC is in low power mode. SIF Slave ASIC will often use a slower internal system clock during low power mode, but the SIF remains unaffected. The SIF Shift phase (on SIF_CLK, SIF_MOSI, SIF_MISO) is still fast. The SIF handshake phase (on SIF_LOADB) automatically slows down to accommodate the slower internal system clock.

Master/pod does not need to be aware of change of mode/clocks/speed in the slave. See SIF Definition document (section 6.6).

In one aspect there is provided a processor interface apparatus, comprising a processor, an interface connected to the processor and comprising means for writing data to and/or reading data from the processor, and control means for controlling operation of the interface.

Preferably the processor is adapted to use a first clock and the control means is adapted to use a second clock. Preferably the interface is adapted to operate such that at least one operation is dependent upon the second clock, and preferably such that the timing of the at least one operation is dependent upon the second clock. Preferably the apparatus is arranged so that a request for the interface to write data to and/or read data from the processor is generated dependent upon the second clock. Preferably the control means is adapted to generate the request.

Preferably the apparatus is arranged so that a read operation for reading data from the processor, or a write operation, for writing data to the processor, is carried out in response to the request, and preferably the apparatus is arranged so that the read operation or the write operation is carried out dependent upon the first clock. Preferably the apparatus is arranged so that the read operation or write operation is commenced dependent upon a time slice process. The time slice process is preferably a time slice process of the processor.

Preferably the apparatus is arranged to carry out the write operation or the read operation independently of the second clock.

Preferably the apparatus comprises means for operating at least part of the interface dependent upon the second clock as well as or instead of the first clock. Preferably the means for operating at least part of the interface dependent upon the second clock is adapted to operate at least part of the interface dependent upon the second clock during at least part of a communication process between the interface and the processor.

Preferably the interface comprises a shift register and preferably comprises a SIF Slave module. Preferably the control means comprises a SIF pod, or master, and/or a control computer. Preferably the processor is a XAP3 processor.

Preferably the interface further comprises means for instructing operation of the processor.

Preferably the processor and the interface are on the same circuit board and/or form part of an ASIC or FPGA. Many of the embodiments described herein relate to an ASIC, but may equally relate to an FPGA.

Features of the preferred embodiment are described, in particular, in sections 2.2, 4.3 to 4.5, and 6.6 of the SIF Definition document.

SIF enables clock speeds to be switched in the slave.

An apparatus is provided which provides asynchronous hand shaking between a master and a slave and which does not change the timing of the application running on, or associated with, the slave.

A memory means associated with, and preferably included in, the processor may be controlled either by the processor or by the interface. The memory means may comprise a memory management unit (MMU).

The MMU may be controlled either by the XAP or by the SIF.
In operation of the processor there may be a time slice process. The position of the time slice within the software program may be set by a processor-sif instruction, or fixed in hardware (like a microcode processor on a fixed loop of, say, 256 instructions, with instruction 0 being a fixed SIF cycle).

Further details concerning communication of instructions from a SIF Pod to a SIF Slave, and communication of data between the SIF Pod and the SIF Slave, and the timing of such communication is provided in sections 4, 6, and 7 of the SIF Definition document .

There is provided a processor-sif instruction and a processor-sleepsif instruction so that a programmer can choose when the SIF instructions (from the SIF Computer and SIF Pod) should be executed.

The processor-sleepsif instruction puts the processor into sleep mode which stops instruction execution until a hardware wakeup input is received. In the case where a processor-sleepsif instruction is used, particularly when there are many sensor tasks, there may be a regular wakeup. The wakeup may occur at the rate at which samples are taken in a data acquisition process, for instance between 1 and 20 kHz.

In the preferred embodiment, communication between the SIF Slave module and the processor may take place, for instance, during a period when the processor is in Sleep mode, or during a processor-sif instruction.

In the preferred embodiment, upon entry into Sleep mode the processor typically stops where it is (a software instruction). In the preferred embodiment, the wakeup is typically a hardware input instructing the processor to re-start.

Software can be organised so that after a wakeup, the processor re-starts and branches to software which vectors to a program location dependent upon the type of wakeup. This is similar to an interrupt process but is simpler.

Typically the processor would get on with its work (for instance data acquisition in one example) when it is awake. That work may take a variable amount of time, as it is often data-dependent. There is then a variable time when the processor is asleep (i.e in sleep mode). In that sleep time, SIF Instructions are carried out. As long as the time taken for the SIF instructions to be carried out is less than the time to the next wake-up then operation is straightforward

SIF activity may take place either in during a processor-sif instruction or during a sleep period (which was initiated by a processor-sleepsif instruction).

For sensor applications, use of sleep and wake up in relation to SIF activity works particularly well. This is because none of the inputs are asynchronous. The whole system timing can be controlled from a single clock source.

For radio systems, or other systems which may not follow a regular pattern of activity, it is necessary to respond to asynchronous inputs. In that case it may be particularly advantageous to organise software in a more interrupt-based manner and to use processor-sif instructions rather than processor-sleepsif instructions to control the when communication between the SIF Slave module and the processor occurs.

## 2. Networked SIF

In a further aspect there is provided the ability to simultaneously monitor and debug a plurality of SIF Slaves. Multiprocessor debugging is provided. Multi-processor support in xIDE (a particular Integrated Development Environment) may be used. Data packets flowing between SIF Slaves (e.g radio or power-line communications devices) may be monitored.

In a further aspect, there is provided a device interface apparatus, comprising a plurality of interfaces each comprising respective means for writing data to and/or reading data from a respective device, and control means for controlling operation of the plurality of interfaces. Each device may be a slave and/or may comprise a processor.

Preferably, the apparatus further comprises a plurality of control means each adapted to control a respective plurality of interfaces.
Preferably each interface further comprises means for instructing operation of a respective device.

Preferably the apparatus further comprises a computer adapted to communicate with, and preferably control, the or each control means. Preferably the SIF computer is adapted to receive data from and/or send data to one or more devices via the or each control means.

Preferably the or each computer is adapted to communicate with a respective plurality of control means using a packet based protocol, for instance TCP/IP or ethernet.

Preferably the apparatus comprises means for detecting errors in packets sent between the or each computer and at least one of the control means.

Preferably, the or each interface comprises a register, preferably a shift register, and preferably the or each interface comprises a SIF slave module.

Preferably the control means comprises a SIF Pod and/or a control computer. Preferably the processor is a XAP3 processor.

Preferably the processor and the interface are on the same circuit board and/or form part of an ASIC or FPGA.

Preferably the interface and/or the control means and/or the computer comprises means for debugging the or each device and/or comprises means for capturing data from the or each device and/or comprises means for analysing data from the or each device.

The figure in section 2.1 of the SIF Definition document shows a plurality of networked SIF slaves linked to a SIF Pod/Master.

Networking of SIF slaves allows a single computer to access many SIF Slaves simultaneously which enables, for instance, de-bugging and/or data acquisition, on a wide scale.

Referring to the figure in section 2.1 of the SIF Definition document, software running on each of the SIF slaves (for instance ASICs or FPGAs) shown can be debugged, because packets can be observed from the computer (shown in purple at the top of the figure) going to each of them.

Multiprocessor de-bug may be operated and/or developed in the xIDE environment.

Debugging of multiprocessors in a single chip may also be provided.

This aspect may have particular application to meshing of networked radio devices, particularly when used for communication of non-voice data which a plurality of separate radio devices are able to identify and communicate with each other. By way of example, meshing may be used to control lights in a warehouse. A particular light may be controllable using a radio transmitter. In the case where the radio transmitter is out of range of the light, the radio

signal from the transmitter may be passed via a plurality of radio devices (for instance located on other lights) located between the light and the transmitter. The radio devices work out, usually upon power up, how to communicate with each other and pass messages on to the intended recipient device.

Each radio device has a unique number. Each device may be used as stand-alone (often battery powered) device, passing data to and via other devices via radio links.

Each device may be connected via Ethernet in order to enable multi-processor debugging. Thus the devices (slaves) may communicate by radio links between themselves but may also each have an Ethernet connection to the SIF Computer to enable debugging and other data transfer.

The SIF may, for instance, be used for debugging and also for production test and data acquisition.

The SIF enables the reading of and writing to various specified addresses or registers inside a chip in real time without changing its functionality or timing. It is non-invasive. In general the timing of the chip is not changed.

This is particularly advantageous in, for instance, DSP systems having a fixed frequency plan and enables communications to take place between a SIF Computer and a SIF Slave, without changing the SIF Slave's timing.

A chip may be connected up in its real mode to, for instance, a sensor, to a radio, to a USB link, to a RS232 link, or to a CAN link and at the same time may have another wire (e.g SIF) going in which may not be part of the application, allowing monitoring of operation without changing operation or timing.

In normal mode when a processor is running, SIF executions take place in a time slice manner.

**3. SIF enables processor functions to be split across devices.**

In a further aspect there is provided a processor apparatus comprising a first processor, a second processor and interface means adapted to enable communication between the first processor and the second processor.

Preferably the first processor and the second processor are adapted to co-operate together via the interface means, and preferably the first processor is adapted to perform first aspects of a process and the second processor is adapted to perform second aspects of a process.

Preferably the second processor is programmable, so that the second aspects of the process may be varied. Preferably the second aspects comprise aspects relating to a user interface, and/or pricing aspects, and/or language aspects (i.e functions that may need to be modified during product lifetime). Preferably the first aspects comprise critical functionality of the process. Preferably the first aspects comprise aspects of the process which require regulatory approval. Preferably the first aspects comprise functionality that is likely to remain fixed throughout the product lifetime.

Preferably the first processor is adapted to carry out the first aspects of the process in dependence upon instructions that may be stored in ROM (because they are likely to remain fixed). Preferably the second processor is adapted to carry out the second aspects in dependence upon instructions stored in RAM, or Flash memory, or in some other rewritable memory (because they are unlikely to remain fixed).

Preferably the interface further comprises means for instructing operation of the first processor and/or the second processor.

Preferably the first processor is part of an ASIC or FPGA. Preferably the interface comprises a SIF interface.

SIF enables silicon partitioning. Two chip solutions to separate fixed operational features from short-term marketing-driven features. This reduces the risk of affecting operation of operational features when altering marketing-driven features. .

There are provided two chips, and preferably alteration of functionality of one chip does not affect functionality of the other chip.

As SIF can be used in real applications, it is easier to do partitioning of functionality between chips. Typically, critical functionality may be provided by one chip and less critical functionality (which may vary for particular products and/or for regulatory or marketing reasons).

A SIF may be used for monitoring and data logging and analysis in, for instance, production, monitoring, and analysis systems

In one embodiment, a SIF is used in a gas meter system. An ASIC is used for front end applications. The ASIC in effect implements all of the critical functionality of the system but different products are made for the German, French, British and American markets due to the different regulatory requirements.

A separate ASIC is not produced for each market. Instead, an external microprocessor is provided which talks to the ASIC over the SIF. The microprocessor can access the ASIC in a way which is not invasive. The design is partitioned so that the critical science functionality cannot be tampered with, as it is locked in the ASIC. Whereas the external microprocessor may vary from one market to another.

In another embodiment, there is separation between features of Bluetooth functionality which have obtained regulatory approval, and features which do not require regulatory approval. A chip is provided which includes features of Bluetooth functionality which require or have obtained regulatory approval. A separate software development environment is provided which can be used

(for instance by a customer) to customise features of the product, for instance, buttons and LED's. It is ensured that in customising features, the Bluetooth functionality cannot be broken, or changed in such a way that it would be needed to reapply for radio approvals.

## 4. Four SIF fields (e.g address, control, data, status)

In a further aspect there is provided a processor interface apparatus comprising a register adapted to be connected to a processor and to read data from and/or write data to the processor. Preferably the register comprises a plurality of fields, preferably at least three fields, each field comprising at least one bit.

Preferably the plurality of fields comprises at least one of, and preferably all of:- an address field, a control field, a data field, and a status field. See SIF Definition document (for instance section 2.2, section 4 , section 7 for description of aspects of the preferred embodiment.

Preferably the address field comprises an address in the processor from which data is to be read or to which data is to be written, and/or the address field comprises information which identifies an operation, for instance a debug operation, to be performed by the processor.

Preferably the data field comprises data to be written to, or data which has been read from, the processor. Preferably the apparatus is adapted to write the data to the address specified in the address field. Preferably the apparatus is adapted to read the data from the address specified in the address field.

Preferably the control field comprises data relating to control of the processor and/or the register and/or a further component of the processor interface apparatus, and/or relating to control of communication between the register and the processor and/or a further component of the processor interface apparatus.

Preferably the control field comprises data relating to at least one of:- error checking, parity checking, processor identification, type of operation to be carried out (for instance read or write), mode information (for instance debug mode, SIF mode), and data-size information.

Preferably the status field comprises data relating to the status of at least one of the processor, the register, and a further component of the processor interface apparatus, and/or relating to communication between at least one of the processor, the register, and a further component of the processor interface apparatus. Preferably the status field comprises data relating to the status of a process carried out by at least one of the processor, the register, and a further component of the processor interface apparatus. Preferably the status field comprises at least one of an error code a parity code and a timestamp. Preferably the contents of the status field are independent of the address.

An error code may comprise a SIF error code or a processor error code. An error code may be included in the status field.

Preferably the register comprises 88 bits. Preferably the address field comprises 32 bits. Preferably the control field comprises 8 bits. Preferably the data field comprises 32 bits. Preferably the status field comprises 16 bits. Such a configuration of the fields is typically used in the XAP3 processor.

Alternatively, the register may comprise 52 bits. In that case, preferably the address field comprises 16 bits. Preferably the control field comprises 8 bits. Preferably the data field comprises 16 bits. Preferably the status field comprises 12 bits.

Preferably the further component of the apparatus comprises at least one of a SIF Pod or SIF Master, communicating with a SIF Computer running SIF Driver software.

Preferably, the register comprises a shift register, and preferably the register forms part of a SIF Slave module. Preferably the SIF Slave module is adapted to communicate with a SIF Pod or SIF Master. Preferably the processor is a XAP3 processor.

Preferably the processor and the register are on the same circuit board and/or form part of an ASIC.

Four SIF fields may be provided. Each of the four fields may be up to 32 bits long. The four fields may be address, control, data and status fields.

The address and data fields for certain embodiments must be 32 bits in length.

Aspects of the preferred embodiment are described, for example, in section 9 of the XAP3 Specification document, and in particular section 9.1 and section 9.4. More aspects of the preferred embodiment are described in section 7.4 of the SIF Definition document.

Address, control, data, and status fields are described in relation to a SIF Slave for one embodiment in section 7.4 of the SIF Definition document.

Section 9.4.5 of the XAP3 Specification document provides an explanation of the nature of each bit in the four SIF fields in the preferred embodiment.

Transfer of data to and from the address, control, data and status fields in the preferred embodiment is illustrated diagrammatically in the SIF ASIC Block Diagram in section 9.1 of the XAP3 Specification document.

An Application Programming Interface (API) is preferably provided.

The API enables a programmer to program an application which talk to SIF Slaves via SIF Pods. The application may, for instance, be written in xIDE or SIF Toolkit or Matlab. The API may be an xSIF API or a SIF API.

The SIF API has the concept of two fields (address, data), each of which can be up to 32 bits, the lengths being parameterisable. This can be used, for

instance, with XAP1 and XAP2 processors. XAP1 may have a 36 bit shift register. XAP2 may have a 64 shift register.

The XAP3 processor requires an 88 bit shift register. This cannot be supported by the SIF API, which can only support shift registers up to 64 bits in length.

The xSIF API has the concept of four fields (address, control, data, status), each of which can be up to 32 bits, the lengths being parameterisable. The xSIF API is capable of supporting the 88-bit shift register used by the XAP3 processor.

It is preferable for the SIF and xSIF APIs to be portable across several different computer platforms and programming languages.

In alternative embodiments, two fields are provided (address & data), each of 64 bits. This can also support the 88-bit shift register used by the XAP3 processor.

In the preferred embodiment, address and data fields have to be 32 bits, and address is interpreted the same way in the SIF as in the processor.

There is provided multi-processor de-bug within a single chip whilst still only having a single SIF. The SIF may have, for instance, four or five pins.

In the preferred embodiment, the XAP3 SIF allows access to up to 8 processors on the same chip.

Preferably in the control field, there is a write bit which identifies whether a SIF instruction is a read or a write.

Preferably in the control field, there is a debug bit which identifies whether a SIF instruction is a normal instruction or a debug instruction.

The same approach may be used for XAP 2.

Preferably in the control field, there are two size bits which say whether a read or write is to use 8,16 or 32-bit data.

## 5. Single Chip (single SIF) support for multiple processors.

Reference is made to section 13 below, entitled Multi Processor SIF support

## 6. Direct SIF instructions for Reset.

In a further aspect there is provided a processor interface apparatus, comprising a processor, an interface and means for a resetting operation of at least part of the apparatus. Aspects of the preferred embodiment are described in the XAP3 Specification (for instance, section 9.7).

Preferably the resetting means is adapted to reset at least part of the processor, at least part of the interface, at least part of at least one further processor, and/or at least part of an associated device. Each of these reset operations can be selected individually or together as a simultaneous group.

Preferably the resetting means is adapted to reset operation of the at least part of the apparatus, preferably the processor or the at least one further processor, independent of the state of operation of the processor or of the at least one further processor.

Alternatively, the resetting means is adapted to reset operation of the at least part of the apparatus, preferably the processor or the at least one further processor, only if the processor or the at least one further processor is in a specified mode, preferably in a debug mode.

Preferably the interface further comprises means for instructing operation of the processor.

Preferably the interface comprises a SIF interface comprising at least one of a SIF Slave module and a SIF Pod. Preferably the processor comprises at least one of a XAP1, XAP2, or XAP3 processor. Preferably the processor forms part of an ASIC or FPGA.

In a further aspect there is provided the following features, alone or in any appropriate combination:- direct SIF debug instruction for reset; RST normal/running; RST debug mode; debug/stopped; options everything off chip; ASIC/SIF selected processor; selected user gates.

Description relating to SIF instructions in the preferred embodiment is provided, in particular, section 9 of the XAP3 Specification document, with particular reference to sections 9.5 and 9.7.

Description of direct SIF debug instructions in the preferred embodiment are provided in section 9.5 of the XAP3 Specification document, and in particular in section 9.5.2 which provides a table of XAP3 SIF instructions. Normal SIF instructions are shown first in the table, followed by direct debug instructions, followed by other debug instructions.

It can be seen from section 9.5.1 of the XAP3 Specification document that in preferred embodiments, SIF instructions are either normal SIF instructions or debug SIF instructions. Debug SIF instructions may be direct debug SIF instructions which are executed in the SIF and do not access any processor, or may be processor debug SIF instructions which refer to a selected processor.

The various reset instructions coming from the SIF may be direct SIF instructions.

Two different styles of reset may be provided, which can differentiate between going into the normal reset mode and going into the reset debug mode. Resets may be to everything, or to a chip, or to specific processors, or to the SIF.

Reset instructions from the SIF give precise control of which bits are reset.

How things are put into and come out of reset is important as a de-bug feature.

## 7. SIF status counters and Direct SIF instructions to read them

In a further aspect there is provided a processor interface apparatus, comprising a processor and an interface, the interface comprising means for reading data from and/or writing data to the processor, and the apparatus further comprising means for monitoring operation of the interface and/or operation of the processor and/or communication between the processor and the interface.

Preferably the interface further comprises means for instructing operation of the processor.

Preferably the monitoring means is adapted to monitor at least one of:- the number and/or type and/or timing of data read or write operations between the interface and the processor; the number and/or type and/or timing of errors occurring in the interface, the processor or in communication between the processor and the interface; the number and/or type and/or timing of reset operations and/or cancellation operations occurring in the interface, the processor, or in communication between the interface and the processor; the quantity of data read, write or transferred; and the timing of data readings by the processor.

Preferably the interface comprises a SIF interface comprising at least one of a SIF Slave module and a SIF Pod. Preferably the processor comprises at least one of a XAP1, XAP2, or XAP3 processor. Preferably the processor forms part of an ASIC or FPGA.

In a further aspect there is provided at least one self monitoring SIF counter. The SIF itself may monitor the number of reads, writes, errors and number of cancellations, preferably using direct SIF instructions.

Description of aspects of the preferred embodiment is provided in section 9, in particular sections 9.5.2 and 9.6.3 of the XAP3 Specification document.

The SIF itself has at least one counter, and preferably four counters, which monitor SIF activity. Preferably the SIF counters monitor at least one of:- the number of SIF reads since reset, the number of writes, the number of errors which have occurred and the numbers of times SIF operation has been cancelled.

This aspect is particularly useful in connection with the running of, say, a data acquisition system, particularly an overnight data acquisition system, gathering large quantities of data. It enables a user to identify if there had been any problems during the data acquisition process.

In the preferred embodiment, there are Direct SIF Instructions in the SIF Slave module to enable a SIF Computer to read the value of these 4 counters via a SIF Pod. Direct SIF instructions are described in more detail in section 9.5.2 of the XAP3 Specification document.

## 8. SIF cancel methods

In a further aspect there is provided a processor interface apparatus comprising an interface and a processor, the interface being adapted to communicate with the processor when the processor is in a communication mode.

Preferably the processor is adapted to enter the communication mode in response to a signal from the interface. Preferably the processor is adapted to exit the communication mode in response to an indication that a communication process has been completed successfully.

Preferably the interface is adapted to communicate with the processor only when the processor is in a communication mode and the interface is in a communication mode. Preferably the interface is adapted to enter the communication mode in response to a signal from the processor that the processor is in the communication mode.

Preferably the interface and the processor are adapted to communicate using a handshake procedure.

Preferably there is provided means for making the processor and/or the interface exit the communication mode after a specified time delay, preferably a specified time delay from entry into the communication mode. Preferably the means for making the processor and/or the interface exit the communication mode is included in a control means.

Preferably the processor is adapted to exit the communication mode in response to a signal, preferably a signal from the interface, regardless of whether the communication process has been completed successfully or not.

Preferably the processor is adapted to exit the communication mode in response to the interface stopping requesting communication. Preferably the

interface is adapted to stop requesting communication in response to a cancel signal from a or the control means.

Preferably the interface comprises a SIF interface comprising at least one of a SIF Slave module and a SIF Pod. Preferably the processor comprises a XAP3 processor. Preferably the processor forms part of an ASIC or FPGA. Preferably the control means comprises a SIF pod and/or a SIF computer.

In a further aspect there is provided at least one SIF cancel method.

Particular features of the preferred embodiment are described at section 4.7 and section 7, of the SIF Definition document, and section 9 of the XAP3 Specification document.

Preferably the master/pod detects SIF lockup with a time out.

The slave may tell the master that it made an illegal request

In known handshake procedures, a master may queue up a request and a slave may be waiting to process the request but may never get an answer and thus may stay stuck low. In that case, a SIF may not be usable thereafter, and may produce an error message such as 'SIF error' (even though the error would not have been with the SIF).

Furthermore, if a number of devices are on the same pod, then if one of them pulls SIF_LOADB low then not only is that device stuck but the pod cannot contact any of the devices because the SIF is being shared by the devices.

Examples of when SIF_LOADB may get stuck low, in known handshake procedures, occur when an invalid request is made, for instance when:- an undefined SIF instruction is given; if it is requested to write to a register which is read-only; a request is made which requires being in debug mode but the SIF is not in debug mode; or a request is made whilst a processor is running

which requires the processor to be stopped or to execute a processor-sif or processor-sleepsif instruction.

The SIF Cancel Method features provide solutions to the problems outlined in the preceding three paragraphs.

Two methods for cancelling SIF operation may be provided.

In the first method, the master pulls the slave chip select pin (SIF_CS) low, causing the slave to no longer be selected and clearing any pending SIF cycles which are stuck. Further details of an embodiment of the first method are provided in section 4.7.1 of the SIF Definition document.

In the second method, the slave automatically stops pulling the relevant pin low, regardless of actions that have taken place, after a certain number of clock cycles, for instance 32 cycles on SIF_CLK. Thus the SIF is cleared simultaneously for all the slaves. Further details of an embodiment of the first method are provided in section 4.7.2 of the SIF Definition document.

The first method typically requires that each slave be cleared in turn. However, usually no more than one slave should selected at any one time, so no more than one slave should be stuck at any one time in any event. The second method typically clears all slaves simultaneously.

Typically if a pod makes a request that can't be completed by the slave because it has not received the relevant processor-sif or processor-sleepsif instructions it leaves SIF_LOADB stuck low. That tells the pod that it has requested something that it shouldn't have done. Typically the SIF Computer detects a timeout and then issues a SIF CANCEL procedure which will clear the stuck SIF and allow SIF_LOADB to go high again.

Further details concerning lock-up and subsequent cancel procedures in the preferred embodiment are provided, for instance, in section 9.3.2.4 of the XAP3 Specification document.

## 9. SIF timestamp. Use of SIF status field

In a further aspect, there is provided a processor interface apparatus comprising a register, preferably a shift register, adapted to be connected to a processor and to read data from and/or write data to the processor.

Preferably the register is adapted to include time stamp data.

Preferably the register comprises a plurality of fields, preferably at least three fields, and more preferably four fields, each field comprising at least one bit, and preferably the time stamp data is included in at least one of the fields.

Preferably the time stamp data corresponds to a portion of data, preferably to a portion of data written to or read from the register. Preferably the time stamp data indicates when the portion of data was written to or read from the register and/or written to or read from the processor. Preferably the portion of data is data from a device, preferably a sensing device, associated with the processor, and the timestamp indicates when the portion of data was recorded by the device.

Preferably the plurality of fields comprises at least one of, and preferably all of:- an address field, a control field, a data field, and a status field.

Preferably the time stamp data is included in the status field. The corresponding portion of data may be included in the data field.

Preferably the register comprises 88 bits. Preferably the address field comprises 32 bits. Preferably the control field comprises 8 bits. Preferably the data field comprises 32 bits. Preferably the status field comprises 16 bits.

Preferably the further component of the apparatus comprises at least one of a SIF Pod or SIF Master together with a SIF Computer running SIF Driver software.

Preferably, the register comprises a shift register, and preferably the register forms part of a SIF Slave module. Preferably the SIF Slave module is adapted to communicate with a SIF Pod, or SIF Master. Preferably the processor is a XAP3 processor.

In a further aspect, a SIF time stamp is provided.

Features of the preferred embodiment are described, for instance, with reference to section 9 of the XAP3 Specification document in particular.

It can be seen in section 9.4.4 of the XAP3 Specification document that the status field is 16 bits in the preferred embodiment, and that the top two bits used are as parity bits (PARITY_S, PARITY_D). STAT is a time stamp, discussed in more detail below.

Processors according to the invention, and in particular XAP processors, may be used, for instance, in software debug, production tests or data acquisition. In preferred embodiments, the SIF is used to acquire data from a processor relating to each of those activities.

In the preferred embodiments, for data acquisition, the master continuously requests data from the slave. The slave attaches a time stamp to each piece of data that it sends to the master. The master can analyse the timestamps to see if it has missed any piece of data, or whether it has received two or more pieces of data from the same sampling period at the master, or whether it has received two or more pieces of data with the same timestamp.

Typically the timestamp is synchronised with a data sampling rate at the master.

Preferably in performing software de-bug, a pod controls transfer of data, and the pod does not have to respond to transactions initiated by the slave.

Preferably, the same SIF interface, with the same pins, is used for debug and for data acquisition. In the preferred embodiment, the data-transfers are always initiated by the SIF Pod and Computer, never by the SIF Slave. This means that in the preferred embodiment the SIF Slave never 'throws' data, even during data-acquisition. The SIF Slave in the preferred embodiment always responds to requests from the SIF Pod, whether it is being used for control or data-acquisition. This makes it much easier to create a deterministic data-transfer system. The problem is that during data-acquisition, some data samples may be acquired more than once, and other samples may be missed. The SIF timestamp is a system solution to this problem.

One example of the use of a data acquisition process is the use of an ADC (analogue to digital converter). Preferably the signal being sampled is over-sampled to ensure that no samples are missed. In characterising an ADC it is important not to miss any samples. It is also important to know whether two copies of the same data have been obtained.

A typical software loop for an ASIC processor comprises:- obtaining new data; then executing the software; then issuing a processor-sleepsif instruction to enable the external SIF Pod to obtain data from the ASIC. In the processor-sleepsif window the SIF can get many reads of the same data. The SIF does not know at that time whether it is in cycle n or cycle n +1.

Preferably there is provided a hardware counter which is incremented at some known rate normally correlated to the data acquisition rate within the chip (for instance about 24kHz).

Preferably the status field is arranged to give extra data which is independent of the address. In the case where a lot of reads are being carried out, the data is from the address requested. The status field is independent of the address requested and can be used for any ASIC register or memory at any address.

Preferably the status field is used to provide a time stamp. In the example given above, the time stamp is an indication of sample number. Accordingly, it

is possible to determine whether duplicate copies of the same data have been obtained, or whether samples have been missed.

In certain embodiments, over sampling take place, and the data obtained may be post-processed, for instance in Matlab. Duplicate copies of data may then be discarded. Accordingly one, and only one, copy of the relevant data is obtained by the post processing.

This feature is particularly useful when using PCs in a data acquisition process as PCs can occasionally miss data acquisition for periods of time, typically 2 to 3 millisecond in length. The use of a time stamp enables identification of periods where data is missing.

Thus 32 bit data can be read, together with a time stamp associated with that 32 bit data.

Considering the time stamp in more detail, shift register bit definitions for the preferred embodiment are provided in, for instance, section 9.4.5 of the XAP3 Specification document.

It can be seen that the shift register bits are divided between address, control, data and status fields in the preferred embodiment, described in section 9.4.5, and that there are 32 address bits, 8 control bits, 32 data bits, and 16 status bits.

Transfer of data to and from the address, control, data and status fields in the preferred embodiment is illustrated diagrammatically in the SIF ASIC Block Diagram in section 9.1 of the XAP3 Specification document.

It can be seen that in the preferred embodiment the 8 bits of the control field are made up of three processor bits (control bits 2:0), one debug bit (control bit 3), two size bits (control bits 5:4), one parity bit (control bit 6), and one write bit (control bit 7).

The bits making up the address, data, and status fields in the preferred embodiment are also described in more detail in section 9.4.5.

It can be seen from the description of shift register bits 40 to 71 (which make up the 32 bit data field) that in the preferred embodiment the data field may be used for 8, 16, or 32 bit read and write operations.

In the preferred embodiment the status field can be used for a variety of purposes. It can be seen that in the example shown in section 9.4.5, two of the status bits are used as parity bits, six of the status bits are used as error bits, and eight of the status bits are used for other status purposes, which may include being used as a time stamp.

In other embodiments, bits, in particular status bits, may be used for other purposes or in different combinations. For instance, thirteen of the sixteen status bits could be used as a time stamp, or for other status purposes.

Preferably, status and control fields are broken down into individual bits, each bit having its own function either alone or in combination with other bits.

Time stamp data may be included in the data field, typically when there is no status field.

The value of the status field is preferably independent of the address field.

The time stamp frequency may be chosen to suit a particular purpose, and may be chosen on command of a user.

Preferably the time stamp is dependent on operation of the SIF and/or of the processor. It may be synchronised with a data acquisition update rate or a software loop rate.

Preferably, the time stamp comprises a count of a number of clock pulses.

Data written to the status field may comprise a software register written to by the processor, as well as or instead of time stamp data.

## 10. Aspects of the processor-sif instruction and SIF instructions

The processor-sif (background debug interface) instruction may be all-1's in 32-bit form, and/or all-0's in 16-bit form. This allows, for example, external SIF control (from the SIF Pod) when the internal ASIC processor data bus is stuck at all-1s or all-0s.

In a further aspect there is provided a processor adapted to execute a debug instruction automatically upon the occurrence of a fault. Preferably the fault is an unintended or unforeseen mode of operation of a computer program being executed by the processor. Preferably the debug instruction is encoded by a binary word corresponding to logic levels that are likely to be found on a bus of the processor following a fault. Preferably the bus is a program memory data bus. Preferably the debug instruction is encoded by a binary word in which all bits are 1or as a binary word in which all bits are 0.

Preferably the processor is capable of executing one or more instructions encoded as a binary word having a first length and one or more instructions encoded as a binary word having a second length, wherein the first length is greater than the second length. Preferably the processor is adapted to execute a long debug instruction encoded as a word of the first length and a short debug instruction encoded as a word of the second length. Preferably the short debug instruction is encoded by a word equal to the one's complement of the least significant bits of the long debug instruction.

In the preferred embodiment, operation of the SIF Slave module may be initiated when a dedicated processor-sif instruction is executed by the processor Core. The processor-sif instruction may be stored in program memory (typically on a flash memory device) and read into the Core via the Memory Management Unit.

In certain embodiments, when a processor-sif instruction is read by the Core, the Core completes execution of its current instruction and sends a signal to the SIF Slave module by setting the Core's "SIF" output signal to a defined

logic-level. The execution of further instructions by the Core is suspended until a "CONTINUE" signal is received by the Core from the SIF Slave module (the Core "CONTINUE" input signal typically using the same logic level as the "CONTINUE" output from the SIF Slave module). By suspending the execution of further instructions by the Core in this manner, the SIF Slave module and debugger can be permitted to have uninhibited access to registers in the Core and to the rest of the apparatus. This can also allow debug access via the SIF Slave module to be easily synchronised with software being executed on the processor.

In a preferred example of a processor which executes 32-bit instructions, the SIF instruction is encoded as 0x00000000. In another preferred example of a processor which executes 32-bit instructions, the SIF instruction is encoded as 0xFFFFFFFF. In other words, the SIF instruction is denoted by an instruction in which all bits are zero and/or in which all bits are 1. Thus, when a SIF instruction is read from program memory into the Core, all of the 32 lines of the program memory data bus are high or all of the 32 lines of the program memory data bus are low. Such an arrangement of logic levels on the program memory data bus is also likely to occur following a fault (such as may be caused by incorrect operation of a program being executed by the Core, for example) or when no software has yet been loaded. Therefore, by encoding the SIF instruction with a binary word that is likely to be found on the program memory data bus following a fault, the processor can be caused automatically to execute a SIF instruction immediately following such a fault. This can allow operation of the SIF module to be initiated in response to the fault thereby allowing software to be debugged.

In a preferred example in which the processor is capable of executing both 16-bit an 32-bit instructions, in 16-bit instruction mode the SIF instruction is encoded as 0x0000 and in 32-bit mode as 0xFFFFFFFF.

Features of the preferred embodiment are described in more detail with reference to, for instance, sections 8 and 9 of the Summary of Inventions document . In particular sections 8.8.8, 9.2, 9.3, 9.5, and 9.7, of the XAP3

Specification document, and with reference to the Additional Information section.

Performance of an operation may require a processor-sif instruction. In certain circumstances, errors may result in the data bus being stuck at all 1s or all 0s.

By making the processor-sif debug instruction all 1s or all 0s, external SIF control, and debugging, is enabled when such errors occur.

Description of direct SIF debug instructions in the preferred embodiment are provided in section 9.5 of the XAP3 Specification document, and in particular in section 9.5.2 which provides a table of XAP3 SIF instructions. Normal SIF instructions are shown first in the table, followed by direct debug instructions, followed by other debug instructions.

As mentioned above, it can be seen from the table in section 9.5.2 of the XAP3 Specification document that in preferred embodiments, SIF instructions are either normal SIF instructions or debug SIF instructions. Debug SIF instructions may be direct debug SIF instructions which are executed in the SIF and do not access any processor, or may be processor debug SIF instructions which refer to a selected processor.
The first three instructions in the table in section 9.5.2 of the XAP3 Specification document, by way of example, are SIF normal instructions, for reading from and writing to memory. The size of the data field, whether 8 bit, 16 bit or 32 bit, is specified. In order to carry out the read or write process, a processor-sif or a processor-sleepsif instruction is first required from the processor.

Some debug instructions may be executed immediately and may not need to wait for a SIF cycle.

In the preferred embodiment, instructions for, for instance, going into debug mode; coming out of debug mode; reading status; reading version number; and reading licence number, may be carried out without being in debug mode.

Once in debug mode, a wide range of actions may be performed, for instance Stop, Run, Single Step, Run to Break.

This breakdown of when SIF instructions are executed in the preferred embodiment is explained better in more detail in section 9.3 of the XAP3 Specification document.

In the normal mode of the preferred embodiment, a SIF operation comprises three stages:- shift in stage, handshake stage and shift out stage. See XAP3 Specification document, section 9.3.

In certain circumstances there may be some ambiguity, as an operation may be validly waiting for something to happen, may be operating with a time delay, or may be stuck (an error having occurred). In each case there may a significant amount of time after the shift in stage, before a handshake occurs, during which time it may not be clear whether the processor is operating normally, and the handshake will occur in due course, or whether the processor has locked up.

In the preferred embodiment, a SIF lock-up may be cleared with SIF Cancel (described in more detail at, for instance, section 9.3.2.4 of the XAP3 Specification document) as discussed in section 8 above.

## 11. 1st SIF clock edge = out, 2nd edge = in

In a further aspect there is provided a processor interface apparatus, comprising a processor, an interface associated with the processor and control means.

Preferably the processor comprises a first clock, and preferably timings of at least some operations, and preferably all operations, of the processor are dependent upon the first clock. Preferably the first clock is a continuously running clock. Preferably the first clock is the ASIC system clock.

Preferably the apparatus comprises means for generating an interface clock signal, and preferably timing of at least some operations of the interface are dependent upon the interface clock signal. In the preferred embodiment, the interface clock signal is SIF_CLK.

Preferably the interface clock signal comprises a series of pulses.

Preferably the interface is adapted to carry out a read operation, preferably a parallel read operation, to read data from the processor. Preferably the interface is adapted to carry out a write operation, preferably a parallel write operation, to write data to the processor. Preferably the read operation and the write operation are carried out by the interface in response to a request from the control means.

Preferably the read operation is carried out dependent upon the interface clock signal.

Preferably the interface comprises a shift register.

Preferably the read operation comprises a load operation comprising loading data into the shift register, preferably from the processor.

Preferably the load operation of the read operation occurs in response to the first edge of the first pulse in the series of pulses of the interface clock signal. Preferably the first edge is a rising edge.

Preferably the write operation comprises writing data from the shift register to the processor, later followed by a load operation comprising loading further data to the shift register.

Preferably the load operation of the write operation occurs in response to the first edge of the first pulse in the series of pulses of the interface clock signal. Preferably the first edge is a rising edge.

Preferably the interface clock signal is held low when no read or write operation is being performed, thus providing lower power consumption.

Preferably the shift register comprises address, control, data, and status fields.

Preferably the data or the further data comprises at least one of address field data, control field data, data field data and status field data.

The load operation may be commenced on a rising edge of a pulse, preferably the first pulse in the series of pulses.

Having the first edge as the shift out edge makes it easier to implement the parallel load mechanism into the shift register in the interface. The reason this is easier is that all of the flip flops in the shift register can use a single ungated clock.

Preferably a load operation commences in response to a rising edge of the interface clock signal.

Preferably the interface comprises a SIF. Preferably the control means comprises a SIF Pod or master and/or a control computer. Preferably the processor comprises an 8bit, 16 bit, 32 bit or 64 bit processor. The processor may be a XAP1 processor, a XAP2 processor or a XAP3 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Preferably the first SIF clock edge is used for parallel load and shift out. Preferably the second edge is used to shift in. This makes the load then shift work properly, ie capture to shift reg (=load) on posedge of SIF_CLK (which makes the msb visible on SIF_MISO), then shift in (to SIF_MOSI) on negedge of SIF_CLK.

There is provided a shift register procedure comprising parallel load first, shift second, using an asynchronous design in which a clock is not always running. Different clock domains are provided.

Features of the preferred embodiment are described with reference, for example, to sections 4, 6 and 7, and in particular sections 4.4.1, 6.2 to 6.6, and 7.2 to 7.6, of the SIF Definition document. Particular reference is made to the figures in sections 4.4.1 and 7.6.

The rising and the falling edge are not symmetrical, one of them is a first.

Data may be clocked into and out of shift registers on opposite edges of a clock signal when coming off devices.

In the preferred embodiment there is provided parallel circuitry inside a slave.

A parallel load may be provided in the SIF Module, and the parallel loaded data may then be sent to the SIF pod here. The parallel load in the SIF Module and the shift out may be on the first edge and the shift in may be on the second edge when doing a load and then a shift.

Clocking in may occur on a negative edge of the interface clock and clocking out may occur on a positive edge of the interface clock and the interface clock may be left low normally.

In the preferred embodiment, operations in the shift register (at the SIF, or SIF Module) are clocked by SIF_CLK, whilst operations in the processor and most of the SIF Slave ASIC are clocked by the system clock. SIF_CLK may be left low, which is a low power state, when not required. The system clock is typically a continuously running clock.

In the preferred embodiment the system clock is used for handshaking on SIF_LOADB. The system clock is typically used to update data capture and the status capture registers and, when complete, SIF_LOADB is allowed to go back to a high state. The Master may then shift out data on the shift register on the SIF_MISO line from the Slave (ASIC or FPGA) to the Master (Pod).

In particular embodiments the contents of certain registers may be transferred to the shift register on the first edge of SIF_CLK.

The system clock may be a continuous running clock. SIF_CLK may comprise a finite series of pulses, for instance 32 pulses. SIF_CLK may be generated externally by the SIF Master (Pod) and is an input to the SIF Slave (ASIC or FPGA).

Parallel load from processor to SIF and serial load from SIF to SIF pod (master) may be provided.

Preferably there is a separation in clock domains between processor and SIF Slave module.

## 12. SIF command mode

The SIF command mode enables plug and play. Different SIFs may be of different length. – The SIF command mode may be used mainly for query functions. Use of the SIF command mode enables query functions without the need to add extra pins, without affecting normal operation and without having any ambiguity of command.

In a further aspect there is provided an interface apparatus comprising an interface, adapted to communicate with a device, for instance a processor, and a control means adapted to communicate with and/or control the interface, the interface being adapted to send data concerning itself to the control means and the control means being adapted to configure itself, or the interface, dependent upon the data.

Preferably the interface is adapted to send the data concerning itself to the control means when in a specified mode, preferably a Command mode. Preferably there is provide a command set for use in the Command mode, the command set comprising a plurality of commands. Preferably the interface is one of a plurality of interfaces, and preferably each interface is adapted to recognise a respective sub-set of the command set or all of the command set. Preferably all of the interfaces recognise a common sub-set of the command set.

By providing such features, interface interrogation functionality is provided without the need to add additional hardware, for instance additional pins, and without affecting existing modes of operation. Existing modes of operation may comprise, for instance, Normal SIF instructions, Debug SIF instructions, and SIF Cancel operations.

Preferably the data comprises at least one of a specific processor or chip identifier, or a generic interface identifier identifying the type of interface, preferably the type of SIF interface, or data representing at least one individual characteristic of at least one component of the interface, preferably

a SIF interface. The at least one individual characteristic may comprise a field length, for instance an address, control, data, or status field length. Also the at least one individual characteristic may comprise information concerning the bits within a field, in particular within the control and/or status fields.

Preferably, the interface may be one of a plurality of interfaces. Preferably each interface comprises a respective shift register. Preferably each shift register comprises a plurality of fields. Preferably, for each shift register,

Preferably, the interface is adapted to send data concerning itself in response to a request signal from the control means. Preferably the request signal comprises a specified sequence of signals. Preferably the apparatus is arranged so that the request signal may be sent from the control means to the interface over a channel which is normally used to send other signals between the control means and the interface, and the interface is adapted to recognise and respond to the request signal when sent over the channel.

Preferably the request signal comprises a specified sequence of 8, 16, 32, 64, 128, or 256 bits or other data units. Preferably the interface is adapted to enter a command mode in response to the request signal. Preferably the interface is adapted to exit the command mode in response to a further request signal, which may be the same as or different to the request signal.

Preferably the interface comprises a SIF. Preferably the control means comprises a SIF Pod and/or a control computer. Preferably the processor comprises an 8bit, 16 bit, 32 bit or 64 bit processor. The processor may be a XAP1 processor, a XAP2 processor or a XAP3 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board. Preferably the request signal represents a command in Command mode.

Features of the preferred embodiment are described with reference to, for instance, section 4, and in particular section 4.8, of the SIF Definition

document, and section 9, in particular sections 9.2 (in particular 9.2.1.2) and 9.3, of the XAP3 Specification document.

SIF normally operates in Normal mode. In the preferred embodiment a Command mode is provided for interrogation purposes, in particular to ascertain characteristics of a particular SIF.

In preferred embodiments, all SIFs recognise certain common commands in Command mode, and all SIFs enter the Command mode in the same way. However, particular SIFs may not be adapted to recognise all commands in Command mode.

Different SIFs may have shift registers of different lengths.

In preferred embodiments, all SIFs recognise request commands requesting a chip identifier, or SIF type, or number or type of processors associated with the SIF, or SIF field length, or information concerning bits within a SIF field.

In preferred embodiments, in Command mode, if the SIF does not recognise a command it generates a response of 0xFF. The range of responses to any valid command in Command mode is 0x00 to 0xFE.

The Command mode enables a SIF plug and play functionality.

SIF characteristics may vary, and in particular characteristics of the SIF shift register or SIF shift register fields, in particular SIF shift register length or SIF shift register field length, may vary for different SIFs and/or for different processors with which a SIF may be associated.

For instance, for XAP1, XAP2, and XAP3 processors the SIF shift register length is typically 32 bits, 64 bits, and 88 bits respectively.

In the preferred embodiment, a command mode is provided. When the SIF is in the command mode, the master can interrogate the SIF, typically using

query functions. When in the command mode, in the preferred embodiment, the SIF does not access the associated processor; the SIF is returned to normal mode when access to the associated processor by the SIF is required.

In the preferred embodiment, the command mode is entered in response to a series of 256 clocks/bits representing successive 'Hex 33333...' signals on the SIF_MOSI line.

In command mode, in the preferred embodiment, all commands are 8 bits in and 8 bits out.

When the SIF is in command mode, the master can ask various questions, for instance 'what is the address field length?' 'what is the control field length?' 'What is the data field length?' 'How many processors do you have?' 'what types of processors are they at the various places?'. There is provided an instruction set which the SIF can recognise in command mode.

Then when you either shift in an instruction of zeros or you pull chip select low or you pull SIF_LOADB low it immediately reverts to normal mode.

Thus the ability to interrogate an interface, or part of an interface, and to set up control means, so as to communicate with and/or control the interface, is provided without adding any extra pins to the SIF and in a way which does not affect normal operation. In the preferred embodiment command mode can be entered without having to pull SIF_LOADB low.

Typically, in the preferred embodiment, when a device is first plugged in, command mode is entered, the device is interrogated, the apparatus is tailored to the device if necessary, and then normal mode is entered. **13.**

## 13. Multi Processor SIF support

Multiple processors per SIF - multi processor support, direct SIF instruction (talking direct to the SIF).

In a further aspect there is provided apparatus comprising an interface adapted to be connected to a plurality of processors, and adapted to communicate with each processor individually. Preferably the interface is also adapted to communicate with the plurality of processors together.

Preferably the interface is responsive to a processor identifier. Preferably each of the processors is responsive to at least one respective processor identifier. Preferably the apparatus is arranged to establish communication between the interface and one or more of the processors in dependence on a processor identifier. Preferably the processor identifier is included in a portion of data, and the interface is adapted to process the portion of data, and in particular to send the portion of data to, or receive the portion of data from, a particular processor or processors, in dependence upon the processor identifier. Preferably the interface is adapted to read data from or write data to a particular processor or processors in dependence upon the device identifier.

Preferably the apparatus is adapted to read the portion of data into and/or out of a register, preferably a shift register.

Preferably the interface is adapted to send a reset instruction to a particular processor or processors identified by the processor identifier. The reset instruction may be one of a plurality of reset instructions. One of the reset instructions may be such as to reset the particular processor or processors to a normal mode of operation, preferably running in the normal mode of operation. Another one of the reset instructions may be such as to reset the particular processor or processors to a debug mode of operation, preferably stopped in the debug mode of operation.

A processor identifier may identify a plurality of the processors, preferably all of the processors.

Preferably the interface comprises a SIF or SIF module, preferably each processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably each processor comprises a XAP1, XAP2, or XAP3 processor. Preferably each interface forms part of an interface apparatus which preferably includes means for debugging, or for acquiring data from, the interface and/or the or each processor. The processors may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processors and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 4, in particular section 4.8, and section 7, in particular section 7.2, of the SIF Definition document, and section 9 of the XAP3 Specification document.

In the preferred embodiment, particular bits may be allocated in every SIF instruction which identify a processor, if any, with which the instruction is concerned.

Typically one SIF per chip is provided, and multiple processors per SIF. Multiple processors per chip may be provided. The chip may comprise an ASIC or FPGA. Each SIF instruction may be directed to any one or ones of the multiple processors.

In the preferred embodiment, there are various ways which allow a master to talk just to the SIF. There are various ways in which the master can find out about the SIF as well as or instead of about a plurality of processors associated with the SIF. For instance command mode enables information concerning the SIF to be obtained. Direct de-bug SIF instructions in normal mode also enable information concerning the SIF to be obtained.

Processor debug SIF instructions may specify one or more of the plurality of processors to which those instructions are to be sent.

By way of example, a particular chip may be connected to a computer. The computer, using a SIF master installed on or connected to the computer, may go into command mode.

In command mode, the SIF master may talk to the SIF to find out how many processors there are on the chip and what types they are. It may read various addresses and subsequently (when having left normal mode) communicate with any of the processors. It may also find out from the SIF what the lengths are of the various SIF fields. It may also obtain an ASIC identifier, if one exists which would tell it more about the chip it is connected to. Subsequently the computer may put the system into normal mode and could then, for instance, debug processor 0 while leaving processor 1 and 2 running. Processor 0 and 1 may be, say, XAP3 whereas processor 2 may be, say, XAP2. Individual processors can be reset or the whole chip can be reset. Single step or run to break point operation may be set on one processor whilst another is set to run normally. All of these functions are provided in the preferred embodiment without adding any extra pins.

In a further aspect there is provided a processor interface apparatus, comprising an interface for writing data to and/or reading data from a plurality of processors, and control means for controlling operation of the interface. Using a single interface to access several processors uses fewer chip pins than having a separate interface for each processor. For example, a single SIF interface with 4 chip pins may be used to access 8 on-chip processors.

Preferably the interface further comprises means for instructing operation of each of the plurality of processors.

Preferably the plurality of processors form part of a single chip. Preferably the plurality of processors form part of an ASIC or FPGA.

Preferably the interface comprises a SIF Slave module. Preferably the control means comprises a SIF Pod and/or a control computer.

Preferably each of the plurality of processors comprises one of a XAP1 processor, a XAP2 processor and a XAP3 processor.

Single chip support for multiple processors is provided. Aspects of this feature in the preferred embodiment are described, for instance, section 9 of the XAP3 Specification document.

## 14. Pipeline SIF reads

In a further aspect there is provided an interface adapted to write data to a device, for instance a processor, in a writing operation and/or read data from the device in a reading operation, the interface being adapted to perform a series of operations, and the interface being further adapted to carry out part of one operation in the series at the same time as carrying out part of at least one other operation in the series, preferably the next operation in the series. Preferably the interface is adapted to operate in pipeline fashion.

Thus, higher data rate operations are enabled than if the interface is operated in non-pipeline fashion.

Preferably the series of operations comprises a series of writing operations and/or a series of reading operations.

The interface preferably comprises a shift register. Preferably the interface is adapted to shift in a part of a set of data from a reading or writing operation in a series of operations at the same time as shifting out a part of a set of data from another reading or writing operation in the series of operations.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the interface forms part of an interface apparatus which preferably includes means for debugging, or for acquiring data from, the interface and/or the or each processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 4, in particular sections 4.4 and 4.5, of the SIF Definition document, and section 9 of the XAP3 Specification document.

In the preferred embodiment the SIF comprises address, control, data and status fields. The SIF may shift in address and control and shift out data and status simultaneously.

In one example using the preferred embodiment, a data acquisition process may require that, say, 10,000 data reads at 12 addresses are carried out. The SIF would operate in a cycle in that example, shifting out data and status data from one read at the same time as shifting in address and control data for the next in a pipeline manner.

## 15. Continuous Shift Reg from SIF_MOSI to SIF_MISO

In a further aspect there is provided an interface apparatus comprising an interface associated with a device, for instance a processor, and a control means adapted to communicate with and/or control the interface, the apparatus being arranged so that data can be sent between the interface and the control means, the apparatus further comprising monitoring means for monitoring changes in data having occurred between sending of the data and receipt of the data.

Preferably the monitoring means is adapted to monitor data received at the interface and/or at the control means. Preferably the interface comprises a shift register, preferably a continuous shift register. Preferably the interface is adapted to suspend communication with the processor during operation of the monitoring means.

There is also provided a method of checking for errors in data transmission or reception in an interface apparatus, preferably an interface apparatus as described herein, comprising sending data between an interface and a control means and monitoring any changes in the data between sending of the data and receipt of the data. Preferably the monitoring step takes place upon receipt of the data at the interface. Alternatively or additionally, the monitoring step may take place upon receipt of the data at the control means.

Preferably the method further comprises sending the data between the control means and at least one further device.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the at least one further device comprises a control computer or a driver installed on a control computer. Preferably the apparatus comprises means for debugging,

or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 5, in particular section 5.3.2, and section 6, in particular section 6.6, of the SIF Definition document, and in section 9 of the XAP3 Specification document

In practice, in the preferred embodiment, data is generally passed between the processor, the SIF, the SIF master, and a computer connected to the SIF master. Data may become corrupted during its passage between those devices, and in particular may become corrupted during its passage over the wires which, in the preferred embodiment, connect the devices.

Using the continuous shift register in the SIF, data can be passed backwards and forwards between the SIF and other devices and checks can be performed to see if data has been corrupted during its passage between devices.

The use of SIF_LOADB is particularly useful in this context as, in the preferred embodiment, when SIF_LOADB is kept high data can still be written to and read from the SIF shift register by the SIF master without there being any effect on SIF slave (see section 6.6 of the SIF Definition document in particular). Thus data can be passed between the SIF and the SIF master, and between the SIF master and other devices if required, and data integrity checks to be carried out, without any effect on the SIF slave.

Continuous shift register from SIF_MOSI to SIF_MISO is useful for data integrity checking. It enables the SIF computer and SIF Pod to rotate data at different rates and discover the maximum rate without data error. It also enables the SIF computer to identify a type of SIF slave from the shift register length.

## 16. Synchronise FIFO with SIF_LOADB for data acquisition

Synchronising a FIFO with SIF_LOADB for data acquisition, without extra pins.

In a further aspect there is provided an interface apparatus comprising an interface and a processor, the processor comprising a register for temporary storage of data, and the interface being adapted to read data stored in the register.

Preferably the apparatus further comprises means for sending a signal to the interface indicating that the register has been updated, the interface being adapted to read updated data stored in the register in response to the update signal.

Preferably the apparatus further comprises control means adapted to communicate with the interface. Preferably the apparatus comprises means for sending to the control means data read from the register by the interface.

Preferably the control means is adapted to establish a communication session with the interface. Preferably the sending means is adapted to send the data from the interface to the control means during a communication session. Preferably the sending means is adapted to maintain a communication session between the control means and the interface pending updated data becoming available to be sent from the interface to the control means.

Preferably the interface comprises an interface register arranged so that data read from the register is read to the interface register. Preferably the register comprises a FIFO (first in, first out).

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the control means

comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Preferably when a master issues a SIF instruction to read the FIFO address the SIF_LOADB handshake will not be completed until the FIFO has been updated with new data and then the processor executes a processor-sif instruction or a processor-sleepsif instruction.

Features of the preferred embodiment are described with reference to, for instance, section 4, in particular section 4.6, and section 5,of the SIF Definition document.

There is provided a FIFO inside a slave. An associated SIF may read data from the FIFO.

A handshake procedure may be provided between the SIF and a master. The master may make a request and the SIF_LOADB may stay low until the new data obtained from the FIFO. The SIF would then release the new data to the master.

## 17. SIF Pod protocol and associated error correction

In a further aspect there is provided apparatus comprising a control means, preferably a SIF pod, adapted to be connected to a control computer, preferably a SIF computer, and to send data to and/or receive data from the control computer, and means for performing an error checking and correction procedure on the data. Preferably the error checking procedure is based upon checking CRC codes included in the data.

Preferably the SIF pod is adapted to be connected to the control computer via an ethernet or USB or RS232 or LPT link.

Features of the preferred embodiment are described with reference to, for instance, section 2, in particular sections 2.1, 2.2, 2.3 and 2.6, and section 5, in particular 5.1.1 and 5.4, of the SIF Definition document.

## 18. SIF error reporting

SIF error reporting - SIF error codes, processor error codes - from pod to slave and/or slave to pod

In a further aspect there is provided an apparatus comprising an interface adapted to communicate with at least one processor, and a control means adapted to communicate with, and/or control operation of, the interface, the apparatus further comprising means for monitoring data received at the or each processor and/or at the interface and/or at the control means, and for detecting an error in the data.

Preferably the monitoring means is adapted to generate an error message upon detection of an error, and preferably is further adapted to send the error message to a source from which the data was sent. The source comprises one of the processor, the interface, or the control means.

The monitoring means is preferably adapted to identify the source of an error, and may be adapted to identify whether the error occurred at the or one of the processors or the interface.

The data may comprise an instruction. The monitoring means may be adapted to detect an error if the processor or the interface or the control means is not in a correct mode to carry out the instruction.

Preferably the data comprises at least one parity bit and the monitoring means is adapted to carry out a parity checking procedure with respect to the or each parity bit, and to detect an error in dependence upon the parity checking procedure.

Preferably the data comprises a plurality of fields, and at least one parity bit is included in or associated with at least one of the plurality of fields. Preferably the plurality of fields comprises at least one of an address field, a data field, a control field, and a status field.

In the case where the error occurs at the interface, the monitoring means is adapted to identify whether the error is a parity error, an unrecognised Direct SIF instruction, an invalid data value, or an unrecognised processor identifier.

In the case where the error occurs at the processor or one of the processors, the monitoring means is adapted to identify whether the error is an unrecognised Normal SIF instruction, an invalid data value, a SIF write to a read only register, a SIF read from a write only register, processor not in debug mode (when it should be for a particular SIF instruction to be recognised), or processor not stopped (when it should be for a particular SIF instruction to be recognised).

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Provision of such features is particularly important in the case of networks of processors and interfaces, where having effective debug functionality is necessary.

Features of the preferred embodiment are described with reference to, for instance, section 5 of the SIF Definition document and section 9, in particular section 9.6, and in particular 9.6.2, of the XAP3 Specification document.

## 19. SIF parity checking

Parity bit checking, in and/or out of the SIF, may be provided.

In a further aspect there is provided an apparatus comprising an interface adapted to communicate with a processor, and a control means adapted to communicate with, and/or control operation of, the interface, the apparatus further comprising monitoring means for monitoring communication of data, preferably between the control means and the interface.

Preferably the monitoring means is adapted to perform an error checking procedure to determine whether there has been an error in communicating data between the interface and the control means.

Preferably the data comprises at least one parity bit and the monitoring means is adapted to carry out a parity checking procedure with respect to the or each parity bit.

Preferably the data comprises a plurality of fields, and at least one parity bit is included in or associated with at least one of the plurality of fields. Preferably the plurality of fields comprise at least one of an address field, a data field, a control field, and a status field.

Preferably the control field contains a parity bit, preferably a PARITY_AC bit. Preferably that parity bit is odd parity (odd number of 1s) for the combined address and control fields.

Preferably the status field contains a parity bit, preferably a PARITY_D bit. Preferably that parity bit is odd parity (odd number of 1s) for the combined data field and the parity bit itself.

Preferably the status field contains a further parity bit, preferably a PARITY_S bit. Preferably that parity bit is odd parity (odd number of 1s) for the status field (including the further parity bit itself).

Preferably the apparatus is adapted to carry out a further operation on the data, or on a field included in the data, in dependence on the outcome of the parity checking procedure.

Preferably the monitoring means is adapted to monitor communication of data between the interface and the processor, and preferably the monitoring means is adapted to perform an error checking procedure to determine whether there has been an error in communicating data between the interface and the processor.

Preferably the monitoring means is adapted to monitor data received by the interface from the control means and/or is adapted to monitor data received by the control means from the interface.

Preferably the interface comprises a SIF, preferably the processor comprises an 8-bit, 16-bit, 32-bit, or 64-bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 5, in particular section 5.3.1, and section 7, in particular sections 7.4.2 and 7.4.4, of the SIF Definition document and section 9, in particular section 9.6, in particular section 9.6.1, of the XAP3 Specification document.

## 20. Serial interface parity checking method that exposes stuck-at faults

In a further aspect there is provided an apparatus comprising means for detecting errors, the error detecting means being adapted to generate at least one parity bit in dependence on the parity of a plurality of subject bits.

Preferably the at least one parity bit is one parity bit.
Preferably the plurality of subject bits includes the or each parity bit.

Preferably the plurality of subject bits is an even number of bits.
In the case where the plurality of subject bits is an even number of bits the error detecting means will generate the at least one parity bit so that the parity of the subject bits and the at least one parity bit together is even (even number of 1s) in the case where the subject bits are all zeros or all ones (which is indicative of an error in preferred embodiments).

In known parity checking systems a single parity bit will only expose a single bit error. In a system according to the present aspect, a word of all 1s or a word of all 0s will also be exposed as an error. This is useful for serial systems as it will expose the serial wire being stuck at 1 and being stuck at 0. In the preferred embodiment, the PARITY_AC bit exposes stuck at faults on SIF_MOSI (see for instance section 9.6.1.1 of the XAP3 Specification document) and the PARITY_S bit exposes stuck at faults on SIF_MISO (see for instance section 9.6.1.3 of the XAP3 Specification document).

Preferably, the error detecting means is adapted to detect errors in a serial operation, preferably a serial read or write operation, in dependence on the at least one parity bit.

Preferably the error detecting means is adapted to indicate a parity error when the subject bits are characteristic of an error having occurred and preferably is

adapted to indicate a parity error when the subject bits are all zeros or all ones.

Preferably the error detecting means is adapted to generate the at least one parity bit so that the parity of the subject bits and the at least one parity bit together is even in the case where the subject bits are characteristic of an error having occurred..

Preferably the apparatus is an interface apparatus, comprising an interface adapted to communicate with a processor and/or a control means. Preferably the error detecting means is adapted to detect errors occurring in the sending of the subject bits between the interface and the control means and/or the processor.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the control means comprises a SIF Pod and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 9, and in particular sections 9.4 to 9.6, of the XAP3 Specification document.

A field may have an even or odd number of bits. Parity checking may be even or odd. Accordingly, there are four possible combinations. Having an even number of bits and having odd parity checking may be particularly advantageous. In that case, in error free operation there will be an odd number of 1s in the field, the field having an even number of bits.

By way of example, in the preferred embodiment a line, for instance the SIF_MOSI line or the SIF_MISO line, could be stuck high or low, and that would cause a parity error.

In the preferred embodiment the PARITY_AC bit is dependent on a section 40 bits long and thus having an even number of bits, and the PARITY_S bit is dependent on a section having 16 bits and thus also having an even number of bits.

In the preferred embodiment, the section on which the PARITY_D bit – which looks at the MISO line going out - is dependent has an odd number of bits. However PARITY_S also looks at the SIF_MISO line coming out and is dependent on a section having an even number of bits.

The PARITY_AC bit in the preferred embodiment looks at the SIF_MOSI line going in.

With reference to the table of section 9.4.5, in the preferred embodiment, the PARITY_AC bit is bit 38 and protects all of bits 0-39. That is, it protects all of the control and address bits.

In the preferred embodiment, the PARITY_S bit is bit 87 and protects all of bits 72 to 87.

If all 0s or all 1s then would get a parity error.

## 21. Two separate reset inputs

RST starts the XAP3 in normal mode and running. RST_DEBUG starts the XAP3 in debug mode and stopped. In a further aspect there is provided apparatus comprising a processor and means for resetting the processor, the resetting means being adapted to reset the processor to one of a plurality of modes. Preferably the plurality of modes comprise a normal mode and a debug mode.

Preferably in the debug mode the normal operation of the processor may be running or may be stopped and/or the processor may be operated a single step at a time in response to debug instructions.

Preferably the resetting means is adapted to reset the processor in response to a reset instruction. Preferably the resetting means is adapted to reset the processor to a normal mode in response to a normal reset instruction. Preferably the resetting means is adapted to reset the processor to a debug mode in response to a debug reset instruction.

Preferably the resetting means is included in, or on the same circuit board or chip as, the processor. Preferably the apparatus includes a dedicated line for communication of the or each reset instruction to the processor. Preferably the apparatus comprises a dedicated line for communication of a normal reset instruction to the processor and a further dedicated line for communication of a debug reset instruction to the processor.

Preferably the apparatus comprises at least one further processor and the resetting means is adapted to reset the at least one further processor. Preferably the resetting means is adapted to reset each of the processor and the at least one further processor by sending a respective reset instruction to that processor.

Preferably the or each reset instruction is specific to a particular processor.

Preferably the interface comprises an interface adapted for communication with the processor and preferably with the at least one further processor. Preferably the interface is adapted to send data to the processor or to the at least one further processor in dependence upon address data, preferably included in the data. The data may include an instruction, for instance the or a reset instruction. Preferably the interface is adapted to receive the data from a control means.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 9, and in particular section 9.7, of the XAP3 Specification document.

## 22. Processor designed for code density – preferably registers not treated symmetrically by the instructions

In a further aspect there is provided a processor instruction set which is not regular, but which is tailored to compiler behaviour and/or calling convention.

Further information is provided, for instance, at section 2.3.1 of the Summary of Inventions documentSummary of Inventions document.

Preferably registers are not all treated symmetrically by instructions. The instructions may play to the coding conventions used by the compiler. In particular instruction set support for the compiler's conventions for function entry and exit can achieve significant improvement in code density.

In the preferred embodiment, certain registers are used for certain functions by the C compiler. For instance there may be certain things that may require to be done by register R5 which are not required to be done with register R6. Accordingly, there may be particular instructions for, say, R5 and not for R6.

In the preferred embodiment the first five function arguments are passed in registers R1 to R5 and subsequent function arguments are passed on the stack (ie in memory). Therefore, in the preferred embodiment there are also provided instructions which have preferential support for registers R1 to R5.

Examples of instructions which have preferential support for R1 to R5 in the preferred embodiment include:- the 16 and 32 bit forms of movm and movm.x (see for instance section 8.10.8.6 of the XAP3 Specification document); the 16 and 32 bit forms of pop (see for instance section 8.10.7.2.1 of the XAP3 Specification document); the 16 and 32 bit forms of pop.ret (see for instance section 8.10.7.2.2 of the XAP3 Specification document); the 16 bit forms of ldm and stm (see for instance section 8.11.9.2 of the XAP3 Specification document); the 16 bit forms of push and push.r (see for instance section 8.11.9.3 of the XAP3 Specification document).

In the preferred embodiment the function return value is passed in register R1 Therefore, in the preferred embodiment there are also provided instructions which have preferential support for register R1.

Examples of such instructions which have preferential support for R1 in the preferred embodiment include:- the 16 and 32 bit forms of pop.ret (see for instance section 8.10.7.2.2 of the XAP3 Specification document).

In preferred embodiments, R14 is used as a link register. This contains the function return address.

In preferred embodiments, R15 is used as a stack pointer. This contains the base address of the stack.

All addresses, in preferred embodiments, are stored in registers as 32 bit byte addresses.

Some instructions may, in practice be used often and other instructions may be used less often. In the preferred embodiments, 16 bit codings are given to those instructions which are used often, and only 32 bit codings are given to those instructions which are used less often.

Preferably there are provided compound instructions, each compound instruction being adapted to carry out a plurality of actions. Thus fewer instruction fetches are required. Examples of such compound instructions in the preferred embodiment include:- movm.*; ldm.*; stm.*; push; push.r; pop; and pop.ret (where * is a wildcard character).

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board or chip.

**23. Instructions may be provided in 32 bit or 16 bit forms.**

Preferably each instruction can be 16 bit or 32-bit, freely mixed on an instruction-by-instruction basis.

16-bit instruction decode may be done serially (16->32) as a decompression stage prior to full 32-bit decode.

In a further aspect there is provided a 16-bit instruction set which is a true subset of a 32-bit instruction set; each 16-bit instruction may be mapped onto a corresponding 32-bit instruction.

In a further aspect there is provided an instruction set having equivalent instructions in short and long forms. Preferably all instructions in the instruction set are available in long form and a sub-set of the instructions is available in short form. Preferably instructions which are used frequently and which can be coded in a short form are provided in a short form.

Being able to mix instructions freely on a long and short form basis means that the processor does not need to have distinct instruction length modes, which thus provides a simple programming model for the programmer.

In a further aspect there is provided a processor adapted to process portions of data of variable length, the processor comprising means for determining the length of a portion of data and being adapted to process the portion of data in dependence upon that determination.

Preferably the determining means is adapted to determine the length of a portion of data in dependence upon a length identifier, the length identifier preferably being included in or associated with the portion of data.

Preferably the processor is adapted to process a series of portions of data in turn, the determining means being adapted to determine the respective length

of each portion of data in turn, and the processor being adapted to process each portion of data in dependence on the respective determined length.

Preferably the or each portion of data comprises an instruction.

Preferably the or each portion of data may comprise at least one operator and may also comprise at least one argument.

Preferably there is provided an instruction set, preferably for a processor as described herein, comprising an instruction provided in a first portion of data having a first length and an instruction provided in a second portion of data having a second length, the first length being greater than the second length. Preferably the first portion of data comprises an argument and the second portion of data comprises a further argument, the length of the argument of the first portion of data being longer than the argument of the second portion of data. Preferably the first portion of data comprises an operator and the second portion of data comprises the same operator. Preferably each of the first portion of data and the second portion of data comprises a respective length identifier identifying the length of that portion of data. Preferably the first portion of data has a length of 32 bits and the second portion has a length of 16 bits.

In a further aspect there is provided an instruction set comprising a first sub-set of instructions which are provided in both a first form and a second form, and a second sub-set of instructions which are provided in a first form. Preferably each instruction in its first form comprises a portion of data of a first length and each instruction in its second form comprises a portion of data of a second length, the first length being greater than the second length. Preferably each instruction comprises an operator and/or an argument. Preferably an or each instruction is provided in the second form or not in dependence on the length of a respective operator and/or in dependence on the length of a respective argument and/or in dependence on the likelihood that the instruction will occur in use of the instruction set. Preferably the instruction set is arranged so that commonly used instructions may be coded

with arguments and/or operators that are sufficiently short that such commonly used instructions may be provided in both or either the first form and the second form. Preferably each instruction in its first form is a 32 bit instruction and each instruction in its second form is a 16 bit instruction.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.2, 8.3, 8.10 and 8.11, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.2 and 2.3.7 of the Summary of Inventions documentSummary of Inventions document.

As can be seen from, for instance, sections 8.2 and 8.3 of the XAP3 Specification document, in the preferred embodiment there are provided both 16 bit instructions and 32 bit instructions. At least some instructions are provided in both 16 bit form and 32 bit form, depending on the length of the argument included with the instruction, such instructions generally providing identical functionality in both their 16 and 32 bit forms.

Commonly used instructions in the preferred embodiment are provided both in 16 bit and 32 bit forms (depending on the length of the argument included in the instruction).

In the preferred embodiment, processor memory includes both 32 bit instruction space and 16 bit instruction space. Typically far more 32 bit instruction space is provided than 16 bit instruction space. For instance there may by, say, 64,000 times more 32 bit instruction space than 16 bit instruction space.

In the preferred embodiment, the processor is defined by a 32 bit instruction set and a subset of that which is used frequently is coded in 16 bits. In the preferred embodiment there are around 32,000 different 16 bit instructions and around 2,000,000,000 different 32 bit instructions.

In the preferred embodiment the 32 bit instruction space overlaps the 16 bit instruction space. Instructions may be identical in their 16 bit form and 32 bit forms but they may they have different (i.e. smaller in the case of the 16 bit form) argument ranges.

Preferably the instruction set is arranged so that roughly 60 to 70% of operations carried out are from 16 bit instructions (with regard to this statement at least, an operation could be considered to be, say, adding 13 to register R5, whilst , say, adding 14 to register R5 would be considered to be a different operation).

In the preferred embodiment the compiler produces assembler code which does not indicate whether an instruction is 16 bit or 32 bit. For each piece of assembler code, the assembler then determines whether there is underlying 16 bit code, or only underlying 32 bit code, available.

Preferably there is provided a 16-32 bit converter adapted to convert 16 bit instructions to 32 bit instructions prior to execution.

As discussed, instructions have a minimum size of either 16 bits or 32 bits in the preferred embodiment. Preferably 16 bit instructions also have a 32 bit form.

Section 8.8 of the XAP3 Specification document shows code numbers allocated to various instructions in the preferred embodiment.

Code numbers may be allocated in a way that makes the decode easier.

In the preferred embodiment, there are 6 bits in the primary op field and there are 64 primary op codes. In the preferred embodiment each primary op code can take a 25 bit argument. The 25 bit argument may be split into different fields.

In the embodiment shown in section 8.8, op codes 56 to 62 are left free. Op code 63 is an extended op code. All extended op code instructions have arguments which are less than 25 bits (arguments of 21 bits or less) in the preferred embodiment.

Details of extended op code instructions in the preferred embodiment are provided in sections 8.8.2 to 8.8.10 of the XAP3 Specification document.

Preferably, some instructions are provided which have a separate prime opcode for each register (eg mov.p). This allows coding long immediates in single instructions.

## 24. Little endian/big endian

In known systems, byte addresses are generally used to address memory for a range of processor sizes (for instance, 8 bit, 16 bit and 32 bit processors).

In known systems, processors include branch instructions to branch to the location of the next instruction to be executed. The address used in the branch instruction is normally the low address of the instruction being branched to (which is significant if the instruction contains more than one byte). That low address is commonly referred to as the low byte. In little endian systems the low byte represents the bottom 8 bits of the instruction (or data word). In big endian systems the low byte represents the top 8 bits of the instruction (or data word).

In a further aspect there is provided a processor having a variety of instruction sizes determined on an instruction by instruction basis.

That enables higher code density than would be achieved with a single fixed instruction size whilst still being simple for the programmer to use.

Preferably each instruction comprises at least one instruction size bit to indicate the instruction size.

Preferably the at least one instruction size bit is provided in the lowest byte, preferably in the lowest bit position, in a little endian architecture processor.

Alternatively, the at least one instruction size bit is provided in the highest byte, preferably in the highest bit position, in a big endian architecture processor.

(Thus it is ensured that the at least one location size bit is always in the byte address branched to. Preferably this is the lowest byte address of the instruction.

Preferably unaligned memory access is supported. A portion of 32 bit data may have any address.

In a further aspect 32 bit instructions and/or 16 bit instructions are provided, and there is also provided a bit included in an instruction which indicates whether the instruction is a 16 bit instruction or a 32 bit instruction.
In a further aspect there is provided a method of processing data comprising a plurality of bits arranged in a series, the plurality of bits comprising at least one bit representing data of a first type and at least one bit representing data of a second type, the at least one bit representing data of a first type being provided in a selected position or range of positions in the series.

Preferably the first type data comprises argument data, preferably at least one immediate. Preferably the second type data comprises operating code. Preferably the data comprises 32 bit or 16 bit data. Preferably the first type data is provided from a particular bit upwards in series of bits. Preferably the first type data is provided from the $8^{th}$ bit (bit 7 when counting from zero) upwards in a series of bits. Alternatively the first type data may be provided from a particular bit downwards in a series of bits (for instance if using a big endian architecture).

Preferably the first type data is processed before, and/or independently of, the second type data.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

In a further aspect there is provided a XAP3 processor adapted to run off the shelf software, for instance Gnu software, such as the Gnu C or C++ compiler, and/or adapted to communicate using TCP/IP protocol, and/or ethernet protocol and/or Bluetooth.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.2 and 8.3, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.1.2 of the Summary of Inventions document.



## 25. 32 bit instruction set characteristics

In a further aspect there is provided an instruction set for a processor, the instruction set having a common format which applies to at least some and preferably all of the instructions in the instruction set.

Thus a simple instruction decode is provided, which leads to low power consumption and a smaller hardware implementation.

Preferably the common format provides a respective fixed position where each of the following register selectors, if present, appears in any instruction to which the common format applies:- a destination register selector (Rd in the preferred embodiment); a primary source register selector (Rs in the preferred embodiment); a secondary source register selector (Rt in the preferred embodiment); a base address register (Ra in the preferred embodiment); and an index address register (Rx).

Preferably the common format provides a fixed position where the least significant bit of an immediate, if present, appears in any instruction to which the common format applies.

In a further aspect there is provided an instruction set for a processor comprising a plurality of instructions, each instruction comprising a respective portion of data and each of at least some of the instructions comprising a respective sub-portion of data comprising a plurality of bits and at least one sign bit.

Preferably at least one of the sub-portions of data is representative of a register selector. In the preferred embodiment the register selector has 4 bits to select one of 16 registers.

Preferably at least one of the sub-portions of data is representative of an argument, preferably an immediate.

Preferably at least one, and preferably all, of the sub-portions of data comprise 17 bits. Preferably at least one, and preferably all, of the sub-portions of data, comprise 16 bits and one sign bit.

Preferably at least one of, and preferably all of, the instructions further comprises an address, preferably a register address, preferably in a sub-portion of data. Preferably at least one of, and preferably all of, the instructions further comprises an operator, preferably in a sub-portion of data.

Preferably at least one of, and preferably all of, the instructions comprise a plurality of sub-portions of data, at least some of the respective sub-portions of data for each instruction being of a fixed length or a multiple of a fixed length. Preferably the fixed length is four bits. Preferably the sub-portions of data for at least some of the instructions are aligned with corresponding sub-portions of data for others of the instructions.

In a further aspect there is provided an instruction set for a processor comprising a plurality of instructions, each instruction comprising a respective plurality of sub-portions of data. Preferably for each instruction at least some, and preferably all of the respective portions of data are of a fixed length or a multiple of a fixed length. Preferably the fixed length is four bits. Preferably the respective sub-portions of data for at least some of the instructions are aligned with corresponding sub-portions of data for others of the instructions.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA.

Preferably, 17-bit immediates are provided, which allow both 17-bit signed (for instance -65536...+65535) and unsigned (for instance 0...131069) value

ranges to be encoded in a single immediate. This enables an implementation with smaller hardware and lower power consumption.

A 4n+1 (where n is an integer) format for the immediates may be provided. Preferably the arguments available in different instructions in the instruction set are of length 4n+1 (where n is an integer), for example, of length 25 bits, 21 bits, 17 bits, or 13 bits.

Preferably processor instructions sign extend or zero extend immediates to represent full word length. In the preferred embodiment, full word length is 32 bits.

A 13 bit argument may be used to represent 3 register selectors plus a carry bit.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.2 and 8.3, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.6 of the Summary of Inventions document.

It can be seen from the tables of sections 8.2 and 8.3 of the XAP3 Specification document that both the 16 bit and 32 bit instructions in the preferred embodiment are aligned in 4 bit sections (nibbles).

In the preferred embodiment, arithmetic and logic instructions are provided in two formats (as described in section 7 of the XAP3 Specification document). In Register, Register, Immediate (RRI) format there is one register input, one immediate input, and one register output. In Register, Register, Register (RRR) format there are two register inputs and one register output.

Typically in the preferred embodiment, particular register addresses occur at the same place in different instructions. For instance it can be seen from section 8.2 that the destination register (Rd) address generally is found at nibble 7. Similarly, each of the first source register (Rs), second source

register (Rt), base address register (Ra) and index register (Rx) addresses are generally found at a respective particular location in the instructions.

In the preferred embodiment, memory is only accessed through a load, from memory to register, or a store, from register to memory, or multiples of such operations, or through push and pop instructions, which are other multi-cycle instructions associated with the stack or with the swap instruction.

Preferably, arithmetic or logic instructions such as AND, OR, ADD, SUBTRACT, MULITPLY do not act on memory. They either take two registers as an input or a register and an immediate and then the output is always a register.

## 26. Move Multiple instructions (movm, movm.x)

In a further aspect there is provided an instruction for a processor, for writing each of a plurality of portions of data to a respective register. Preferably at least one of the portions of data comprises an immediate.

Preferably the instruction comprises a plurality of arguments, each argument comprising a respective one of the portions of data or a respective address (preferably a register selector).

Preferably there is associated with each portion of data a respective flag which indicates whether that portion of data should be interpreted as an address (for instance a register selector) or as an immediate.

Preferably an immediate value of 0 is interpreted to represent -1. This is possible in the case where a register (in the preferred embodiment R0) represents zero. Thus code density may be further improved.

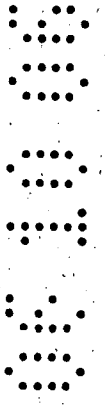
Preferably the or each address comprises the address (preferably a register selector) where a respective portion of data is stored, and preferably the instruction, when performed, is such as to cause the portion of data stored at the or each address (preferably a register selector) to be written to a respective one of the registers. Preferably the or each address is a register address or identifier, such as a register selector. Preferably the instruction comprises a series of arguments and the instruction, when performed, is such as to cause each of the series of arguments, or a respective portion of data obtained by processing of each of the series of arguments, to be written, in order, to a respective one of a series of registers.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.10.8.6 and 8.11.7.1, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.3.2 of the Summary of Inventions document.

In the preferred embodiment, the movm instruction can move up to five registers or small immediate values into R1-R5, thus replacing up to five mov instructions with a single movm instruction.

In the preferred embodiment, the order of writing to registers for movm is from R1 to R5. In contrast, the order of writing to registers for movm.x is from R5 to R1. . By providing both movm and movm.x, chained copies of registers is possible from low to high registers and from high to low registers.



## 27. Instruction mnemonics

In a further aspect there is provided a set of instructions for a processor, each instruction having a respective mnemonic, the mnemonics being organised hierarchically.

Such hierarchical organisation makes it easier for a programmer to learn and use the set of instructions.

Preferably the set of instructions comprises a set of assembler instructions. Preferably each of the set of instructions comprises a respective series of components separated by separators. Preferably each of the series of components is ordered by the significance of each component. Preferably each separator is a dot. Preferably the mnemonics are case sensitive. Preferably the mnemonics are all lower case.

Preferably the instruction mnemonics are of the form base.parameters.type.size.

Preferably the format of assembler identifiers is such as not to restrict the names available in C source code. For example, in the preferred embodiment, %r1 and %sp are assembler identifiers, and they could not in any event be used in C.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA.

Features of the preferred embodiment are described with reference to, for instance, section 2, and in particular section 2.5, and section 8, in particular section 8.4, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.4.2 of the Summary of Inventions document.

In the preferred embodiment the instruction mnemonics are of the form base.parameters.type.size. All mnemonics include base. Parameters, type and size are optional but where they exist they are in the order base.parameters.type.size. Examples of such instruction mnemonics include:-

add.i
add.h
add.r
add.c.r
cmp.c.r
cmp.8c.r
cmp.16xc.r
or.i
or.h
or.r
bra.a
bra.a.4
bra.a.2
bra.p

In the preferred embodiment there is no instruction name space pollution. Names used in C do not need to be converted as they go through the assembler.

Preferably reserved word features used in assembler would not be valid in C. Preferably any reference to a register or registers is prefixed by a percentage. Preferably immediates are prefixed with a hash. Preferably the assembler instructions are case sensitive. Preferably reserved words in assembler are lower case.

## 28. High immediates and down-extending

In a further aspect there is provided an instruction which allows a programmer to explicitly specify the value of the high bits in an immediate, and which provides that all unspecified low bits take the value of the least significant specified bit. This is effectively down-extending the least significant specified bit of the immediate.

In a further aspect there is provided an instruction for a processor, for performing an operation dependent on a first portion of data, the operation comprising setting the value or sign of a second portion of data dependent on the value of at least part of the first portion of data, or generating the second portion of data to have a value or sign dependent on the value of sign of at least part of the first portion of data.

Preferably the first portion of data is included in the instruction. Preferably the first portion of data comprise an immediate.

Preferably the operation comprises setting each bit of the second portion of data to be 0 or to be 1 or to be the same as a selected bit in the first portion of data. Preferably the selected bit is the lowest bit in the first portion of data.

Preferably the instruction may be referenced by an instruction reference, which is recognised by a processor, the instruction including a base instruction reference and an additional reference. Preferably the processor recognises the base instruction reference as being to a further instruction. Preferably the additional reference comprises a suffix to the base reference, and preferably comprises a type mnemonic, and preferably comprises .h, .i, or .r.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA.

Features of the preferred embodiment are described with reference to, for instance, section 2, and in particular section 2.5, more particularly section 2.5.2.6, and section 8, in particular section 8.8.1, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.12 of the Summary of Inventions document.

In the preferred embodiment, the high-immediate ".h" instructions combined with the 17-bit immediate feature support the setting of the upper half of a 32-bit value to a 16-bit value together with setting the lower sixteen bits to 0 or 1. This is particularly useful in creating bit masks where the remaining bits of a 32-bti mask must be 0 or 1. ".h" instructions combine two instructions (setting the upper and lower halves of a 32-bit value) into a single instruction.

In the preferred embodiment, a single immediate instruction can be used to assign a full 32 bit immediate to a SIF where either all the top bits are all zeros or all ones or all the bottom bits are all zeros or all ones.

Thus, for example, in the case where it is only desired to do work in the top (or bottom) 16 bits, only one instruction rather than two is needed to set the other bits to an appropriate value. In the preferred embodiment, values are extended downwards.

Examples of .h and .i instructions are listed in section 8.8.1 of the XAP3 Specification document. It can be seen, for example, that instructions 32 to 37 are all .h instructions, and that instructions 38 to 46 are all .i instructions which all have the low immediate registry. In the preferred embodiment, the nomenclature used, after an index, is u for unsigned, s for signed and h for high (extending downwards).

## 29. push and push.r instructions

In a further aspect there is provided an instruction (for instance push and push.r in the preferred embodiment) which writes multiple registers to a stack, allocates space for local (automatic) variables, and updates a stack pointer register accordingly.

In the preferred embodiment, the push instruction pushes all selected registers from R1 to R14 onto the stack (see for instance 8.10.7.1.1 of the XAP3 Specification document). This saves many instructions at the beginning of a function for saving call-save registers on the stack. This reduces power consumption.

In the preferred embodiment, the push.r instruction pushes registers from R6 upwards, and then copies any of registers R1-R5 to matching registers from R6 onwards (see for instance 8.10.7.1.2 of the XAP3 Specification document). This saves many instructions at the beginning of a function for saving call-save registers on the stack and copying argument registers into the upper registers. This uses even fewer memory accesses than the push instruction, thereby further reducing power consumption.

In the preferred embodiment, both the push and push.r instructions assume that R15 is the stack pointer.

Efficient function entry and exit is essential for good code density. In the preferred embodiment, efficient function entry is provided by the push and push.r instructions.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, or XAP3 processor. The processor may form part of an ASIC or FPGA.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.10.7.1, of the XAP3 Specification

document. Further information is also provided, for instance, in section 2.3.3.3 of the Summary of Inventions document.

## 30. pop, pop.ret instructions

In a further aspect there is provided an instruction for a processor (for instance a pop or pop.ret instruction in the preferred embodiment) which loads at least one register from memory, via a stack pointer, removes zero or more words of automatic storage (for instance as allocated by an instruction according to section 29 above, such as a push or push.r instruction in the preferred embodiment), and updates the stack pointer.

Preferably the instruction takes an additional argument (preferably a register or small immediate) and stores that argument in a register (for instance in R1 in the preferred embodiment), and updates condition flags as if that value was compared with 0. Preferably the instruction (for instance the pop.ret instruction in the preferred embodiment) then executes a return via another register (for instance the Link Register in the preferred embodiment). This reduces the number of instructions, thus reducing memory size and power consumption.

Efficient function entry and exit is essential for good code density. In the preferred embodiment, efficient function exit is provided by the pop and pop.ret instructions.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.10.7.2, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.3.4 of the Summary of Inventions document.

## 31. Load Multiple (ldm) / Store Multiple (stm) instructions

In a further aspect there is provided an instruction for a processor to copy each of a plurality of registers to a respective one of a plurality of memory locations. Preferably the plurality of memory locations are contiguous. Preferably each of the plurality of registers is represented by a respective single bit in the instruction.

In the preferred embodiment this instruction is called store multiple (stm), and is described for instance in section 8.10.6.3 of the XAP3 Specification document. P

In a further aspect there is provided an instruction for a processor to copy data stored at each of a plurality of specified memory locations to a respective one of a plurality of registers. Preferably the plurality of memory locations are contiguous. Preferably each of the plurality of registers is represented by a respective single bit in the instruction.

In the preferred embodiment this instruction is called load multiple (ldm), and is described for instance in section 8.10.6.2 of the XAP3 Specification document.

In the preferred embodiment a single ldm instruction replaces up to 16 load instructions, thereby reducing memory and power consumption.

In the preferred embodiment a single stm instruction replaces up to 16 store instructions, thereby reducing memory and power consumption.

In the preferred embodiment, R0 represents the flags register.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.10.6, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.3.1 of the Summary of Inventions document.

## 32. Movm arguments encoded as 5 bit fields

In a further aspect there is provided an instruction for a processor which moves each of a plurality of data elements into a respective one of a plurality of registers, the instruction comprising at least one identifier and a plurality of arguments, each argument being representative of a respective one of the plurality of data elements, the at least one identifier identifying, for each argument, the type of data of which that argument is representative. Preferably each argument, when processed, is processed in dependence upon the type of data of which it is representative. Preferably the types of data comprise register data and immediates, and each argument comprises one of a register address and an immediate value.

In the preferred embodiment, both movm source and pop.ret return value fields can be registers or immediates, encoded as a 5-bit field: one bit specifies register/immediate, the remaining four bits specifying either one of R0-R15, or an immediate value of –1, 1, ..., 15.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.2 and 8.10.8.6, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.4 of the Summary of Inventions document.

## 33. Register R0 = constant 0

In a further aspect there is provided a processor having a register set to a particular value, preferably zero. There is also provided a method of programming the processor comprising referring to the register rather than to the particular value. There is also provided a set of instructions for the processor which includes references to the register rather than to the particular value. There is also provided a set of instructions for the processor in which no instruction to specify an immediate having the particular value is provided, but an instruction which refers to the value of the register is provided.

In the preferred embodiment, in general, instructions may refer to R0 rather than specifying an immediate with a value of 0.

In the preferred embodiment having register R0 = 0 allows #0 (immediate) to mean other things for decode.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.2, and section 5, in particular section 5.4, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.5 of the Summary of Inventions document.

## 34. Uneven register visibility in instruction set

In a further aspect uneven register visibility in instructions, for instance in 16 bit instructions is provided to optimise for code density.

In a further aspect there is provided a processor and associated instructions, in which registers of the processor are not all treated symmetrically by the processor instructions. Preferably the use of registers is tailored to the coding conventions used by the compiler. Certain registers are used for certain functions by the compiler.

In the preferred embodiment, the design of the 16-bit instruction set has been further optimised to be even closer to the output of the compiler; for example, some 16-bit instructions allow access to R1-R4 and R6-R9, skipping R5 due to register usage behaviour of the compiler.

The preferred embodiment has been designed for code-density.

There are certain things that it may be desired to do with, say, register R5 which it is not desired to do with, say, register R6. Accordingly, in the preferred embodiment there are certain instructions for R5 and not for R6.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.3 and 8.11 of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.8 of the Summary of Inventions document.

## 35. sleepsif and sleepnop instructions enable external debugging

In a further aspect there is provided an instruction for a processor adapted to communicate via an interface, the instruction when processed by the processor causing communication from the interface to the processor and/or from the processor to the interface to be stopped or not allowed, preferably for a fixed period and/or until a further instruction is processed.

In a further aspect there is provided an interface for communication with a processor, the interface being responsive to an instruction from the processor to stop communication with the processor and/or to not allow communication with the processor and/or to not allow transmission of data from the interface to the processor, preferably for a fixed period and/or until receipt of a further instruction.

A sleepsif instruction may allow external debugger access to processor registers and memory while the processor is in the low-power sleep state. A sleepnop instruction may prevent SIF accesses. These two instructions allow an application programmer to control precisely when a SIF has access to the system's resources.

Distinct sleepnop and sleepsif instructions may be provided. Control from the slave end is thus provided, enabling it to ensure that the SIF is quiet, for instance allowing the slave to take readings.

Preferably the processor has two states: a "sleeping" state and an "awake" state. In the sleeping state at least part of the operation of the processor may be suspended, for example by suspending the execution of any further instructions, which may reduce the power consumed by the processor. In the awake state, the processor may be capable of a greater degree of operation than is possible in the sleeping state: for example the processor may execute instructions that are not available in the sleeping state. Thus, the greater degree of activity permitted in the awake state results in the processor

consuming more power whilst in the awake state than whilst in the sleeping state.

The sleeping state may comprise two sub-states: a sleepsif state and a sleepnop state. In the sleepsif state the processor may be capable of performing functions related to debugging, and in particular a SIF Module may be fully operable. However, in the sleepsif state, other functionality of the processor may not operable and in particular the processor may not operable to execute instructions unrelated to debugging such as those of User Mode programs.

In the sleepnop state, the processor may not be operable to execute any instructions: more particularly, neither instructions related to debugging nor instructions unrelated to debugging may be executed in the sleepnop state. Thus, in the sleepnop state, the SIF Module may be inoperable. The provision of the sleepnop state can further reduce the power consumption of the processor relative to that of the sleepsif state and the processor awake state. Additionally, by deactivating the SIF Module in the sleepnop state, electrical noise that may be produced by the SIF Module may be reduced, which may allow more accurate measurements to be made by an analogue to digital converter that is incorporated in an electrical circuit comprising the microprocessor.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.4 and 8.10.14, and section 9, in particular section 9.3, of the XAP3 Specification document, and in the SIF Definition document. Further information is also provided, for instance, in section 2.3.9 of the Summary of Inventions document.

## 36. Support of position independent code

Support of position-independent code (PIC) requires computing addresses from some defined reference point. In a further aspect there are provided two reference points: a Program Counter (PC) for code and constant data references, and a Global Pointer (GP) for variable data references.

In the preferred embodiment the compiler generates PC relative code and constants and GP relative global variables. Thus providing position independent executables and data.

In a further aspect there is provided an instruction set for computing absolute addresses at run time from the PC relative and GP relative instructions.

In the preferred embodiment the PC relative instruction is mov.p (see for instance section 8.10.8.1 of the XAP3 Specification document). Other PC relative instructions include bra.p and bsr.p (see for instance section 8.10.1.2 of the XAP3 Specification document) and all conditional branch instructions (see for instance section 8.10.2 of the XAP3 Specification document).

In the preferred embodiment the GP relative instruction is mov.g (see for instance section 8.10.8.2 of the XAP3 Specification document).

This means that if multiple instances of a program are required to run simultaneously, only one set of code and constants need be stored in memory. Preferably, separate copies of memory are used for global variables for each instance.

These instructions minimise run time fixups for position independent code thereby reducing load time and power consumption.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.10.8.1 and 8.10.8.2, of the XAP3

Specification document. Further information is also provided, for instance, in section 2.3.10 of the Summary of Inventions document.

## 37. Conditional Set instructions

Many expressions in C compute some value dependent on the result of some test.

In a further aspect there are provided sCC.* instructions (where "CC" indicates a given condition code) to support this common code pattern, with two variants. Preferably the sCC instruction assigns to the output register either one of two registers (".r" form) or one of a register and an immediate value (".i" form), based on the state of the processor's condition code register.

Both forms are advantageous as they combine a conditional branch and a move into a single instruction. This reduces memory size and power consumption.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.10.3, of the XAP3 Specification document. Further information is also provided, for instance, in section 2.3.11 of the Summary of Inventions document.

## 38. Breakpoint debug support

Processor code may have debug registers with a count conditional to trigger a breakpoint after some numbers of executions of the target instructions. This simplifies debugging of say, loops.

As this is part of the core, and not an external module, it facilitates common debug environment both for development and in-field debugging. Especially for debugging user mode code by an operating system.

There may be provided hardware breakpoint registers in the processor core which support software-implemented multi-stage breakpoint. The breakpoint registers may use in-core debug registers, and/or a debug module within OS, and/or a textual or graphical interface to specify the breakpoint state.

There is provided a SIF-Ethernet pod that allows for remote debugging of applications over the internet.

In a further aspect there is provided a processor core having a debugging means.

Preferably the debugging means is integral with the core. Preferably the debugging means is fully contained within the core.

A further aspect provides a debugging means. A yet further aspect provides a debugging means for a processor core.

Preferably the core is adapted to execute a computer program. Preferably the debugging means is adapted to facilitate the debugging of a computer program. For example, the debugging means may trigger a predetermined instruction (or sequence of instructions) upon the occurrence of one or more specified events in the program to be debugged, thereby enabling the operation of the computer program to be monitored by a programmer or engineer.

Preferably the debugging means is adapted to enable the execution of a debug processing step. For example, execution of the debug processing step may include the execution of one or more instructions, subroutines and/or computer programs.

Preferably the debugging means includes a breakpoint address register. Preferably the breakpoint register is adapted to reference (or to store information relating to) the address of a memory location. Preferably the breakpoint register is adapted to reference a memory location external to the processor core.

Preferably the debugging means is adapted to enable the execution of the debug processing step when predetermined processing is performed in relation to the address referenced by the breakpoint address register.

Preferably the predetermined processing includes one or more of:

i) the address referenced by the breakpoint address register being accessed a predetermined number of times

ii) a predetermined data value (or a range of predetermined data values) being read from and/or written to the address referenced by the breakpoint address register.

Preferably the debugging means is adapted to enable the execution of the debug processing step when the address referenced by the breakpoint address register has been accessed a predetermined number of times.

Preferably the debugging means is adapted to enable the execution of the debug processing step when the address referenced by the breakpoint address register has been accessed a predetermined number of times by a selected access mode. Preferably the access mode is selectable from the following: write (relating to the writing of data to memory); read (relating to the reading of data from memory); or fetch (relating to the reading of an instruction from memory).

Preferably the debugging means comprises a breakpoint control register for storing information relating to the selected access mode.

Preferably the debugging means comprises a count register for storing information relating to the number of times that the address referenced by the breakpoint address register has been accessed.

Preferably the debugging means is adapted to enable the execution of the debug processing step when a predetermined data value (or a range of predetermined data values) is read from and/or written to the address referenced by the breakpoint address register.

Preferably the debug processing step is selectable from a plurality of such debug processing steps.

Preferably the debug processing steps include one or more of the following:

i) generating an exception

ii) halting the processor.

Preferably the processor comprises a break enable flag for storing information relating to the selected debug processing step.

Preferably the core comprises a plurality of operating modes. Preferably the operating modes are selectable. Preferably each of the operating modes has a respective privilege level. For example, the core may have one or more of the following operating modes: a privileged Supervisor Mode; a privileged Interrupt Mode; a privileged NMI (non-maskable interrupt) mode; and an unprivileged User Mode.

Preferably the debugging means comprises a register that is not accessible by a program operating in an unprivileged operating mode. Preferably the debugging means comprises a register to which data cannot be written by a program operating in an unprivileged operating mode. Preferably the

debugging means comprises a register from which data cannot be read by a program operating in an unprivileged operating mode.

A software debugger may be used to debug a program being executed on a processor that does not comprise a hardware debugger, or that comprises a hardware debugger without this functionality. In the case of a software debugger, the registers and/or flags described herein may be implemented in software as variables.

Features of the preferred embodiment are described with reference to, for instance, section 5, and in particular section 5.5, and section 9, of the XAP3 Specification document.

Certain features of the preferred embodiment, and of possible alternative embodiments are now described in more detail, purely by way of example.

In the preferred embodiment, the processor executes a sequence of instructions comprising one or more programs. Such programs require testing, inspection and verification during development, and testing and correction when the product is in use

The use of multiple processing units in a single system, or even a single integrated circuit, requires the support of an integrated and uniform debug and verification system to make it feasible to develop reliable and robust software within acceptable timescales.

In many applications of the processor the program is held in memory which is difficult or impossible to change once the product has been assembled.

A means is described in which the processor core itself can stop and examine the state of one or more programs currently executing on the processor. The invention is part of the core of the processor, and as such is present in all variations of the processor core. This facilitates a debug infrastructure (debug software, external debug interface, remote debug software, communication

protocols, etc) common to all instances of the processor. This reduces development costs through design re-use, and easily supports the debugging of multi-processor designs through a common debug interface shared among the processor cores.

A processor includes a set of registers (the processor interface) and supporting hardware gates (the implementation).

Certain registers, known as breakpoint registers, can contain information relating to breakpoints. A breakpoint is a point of execution of a program that is characterised by either the address of an instruction in the program (or one program of several), or the address of a location in memory at which the program may access. Upon encountering a breakpoint execution may stop and control of the processor be taken by other software (e.g. operating system, debug executive, external debugger, etc).

These breakpoints may additionally be qualified by either or both of two predicates (conditions):

a) a count of the number of times the instruction may execute, or memory be accessed, before causing a break;
b) a predetermined data value (or one or more ranges of predetermined data values) transferred to or from memory external to the processor core.

In the preferred embodiment, the processor comprises a debug system supporting eight breakpoints, four of which are further qualified with a count predicate, and in which two of these count-qualified breakpoints are yet further qualified by data predicates.

In the preferred embodiment sixteen breakpoint registers and a breakpoint control register to describe the behaviour of eight breakpoint are provided. The sixteen breakpoint registers are named BRK0 through BRK7, BRKCOUNT4 through BRKCOUNT7, BRK6DATA, BRK6MASK, BRK7DATA and BRK7MASK. The breakpoint control register is named BRKE. A single

bit in the processor's FLAGS register specifies if breakpoints should occur or be ignored.

The breakpoint address registers, BRK0 through BRK7, respectively describe a 32-bit wide address for each of the eight breakpoints. These addresses can either be the address of an instruction or the address of a unique memory element external to the processor core.

The count registers, BRKCOUNT4 through BRKCOUNT7, respectively describe a 32-bit wide count predicate corresponding to breakpoints 4 through 7 (as specified in the registers BRK4 through BRK7). A break occurs if an address match happens (that is, the address specified in a breakpoint address register matches the address to/from which an instruction or data is to be read, written or stored) and if the value of the count register corresponding to that breakpoint address register is zero. If an address match occurs, but the value of the corresponding count register is not zero, the value of the count register is decremented by one and no break occurs. The count register preferably cannot be decremented below zero: once it reaches zero, it remains at zero until a new value is written to it.

The default value of the count registers is zero, such that normal behaviour is the same as for the first four breakpoints. That is, when the value in a count register is zero, the corresponding breakpoint behaves like a simple breakpoint.

The BRK6DATA and BRK6MASK registers further specify a data qualifier on the corresponding breakpoint, as specified in register BRK6. These registers operate such that a break occurs when a data value read from (or written to) memory at the location specified in register BRK6 satisfies the data qualifier. It will be appreciated that the "data value" can preferably be either an instruction or data. The data qualifier is specified for each of the 32 bits in the data value. For each bit, the qualifier is satisfied if either

a) the corresponding bit in the mask register, BRK6MASK, is a '1' and the corresponding bit in the data register, BRK6DATA, matches (that is, is equal to) the corresponding bit in the data value, or
b) the corresponding bit in the mask register is '0'.

Thus, by setting a bit in the mask register to '1', the occurrence of a break is conditional upon the bit of the data value that corresponds to that bit of the mask register being equal to the corresponding bit of the data register. Alternatively, by setting a bit in the mask register to '0', a "don't care" condition is selected, and the occurrence of a break is not conditional upon the value of the corresponding bit of the data value.

The BRK7DATA and BRK7MASK registers operate similarly to the BRK6DATA and BRK6MASK registers, but in respect of the breakpoint specified in register BRK7.

Thus, a break can be caused when a predetermined data value is read from and/or written to a predetermined location in memory. By setting a plurality of bits of the mask register to '0', a break can be caused when a data value belonging to a set of predetermined data values is read from and/or written to memory. For particular values of the mask register, the set of predetermined data values can include one or more ranges of consecutive values.

The default value of the mask and data registers is zero, such that normal behaviour is the same as for the first six breakpoints. That is, when a mask register is set to zero, the corresponding breakpoint behaves like a simple breakpoint, since all bits of the mask register treated as "don't cares" and the breakpoint will occur irrespective of the data value being read from and/or written to memory.

The breakpoint control register BRKE is a 32-bit wide register. It is split into eight 4-bit wide fields, where each field corresponds to one of the eight breakpoints.

Each field has three control bits and one unused bit; the fourth bit is left for future debug facilities, for example. The three control bits specify whether a breakpoint should occur:

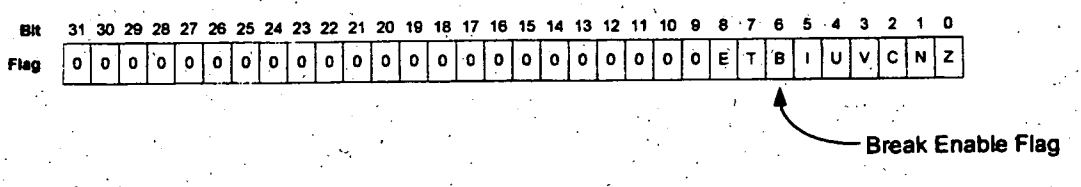
a) if an instruction writes to the address matching the corresponding breakpoint address;
b) if an instruction reads from the address matching the corresponding breakpoint address;
c) if an instruction has been fetched for execution from the address matching the corresponding breakpoint address.

Finally, there is a breakpoint enable bit in the FLAGS register. If this bit is set and the processor is in the lowest privileged execution mode then breakpoints will cause an exception to occur to allow the debug software to take control of the processor and monitor and/or modify the state of the program or programs. Thus, by setting the breakpoint enable bit, an exception is generated rather than halting the processor.

Features of a debug system for a processor will now be described in detail.

**Flags register**

The FLAGS register describes or indicates various signals within the processor core. The 'B' flag controls whether breakpoints cause breaks in execution or are ignored. The diagram below indicates the position of the 'B' flag within the FLAGS register.




**BRKE Register**

The BRKE register is used to control the behaviour of the eight hardware breakpoints described above.

The bits in the BRKE register are described in the table below.

| *Bit(s)* | *Flag(s)* | *Name* | *Description* |
|---|---|---|---|
| 0, 4, 8, 12, 16, 20, 24, 28 | W0, W1, W2, W3, W4, W5, W6, W7 | Write | If set, an instruction making a memory write to the address matching the address (all 32 bits) will halt the processor after the instruction has executed. The Program Counter (PC) will point at the next instruction to be executed. |
| 1, 5, 9, 13, 17, 21, 25, 29 | R0, R1, R2, R3, R4, R5, R6, R7 | Read | Similar to above, but for memory reads. |
| 2, 6, 10, 14, 18, 22, 26, 30 | E0, E1, E2, E3, E4, R5, E6, E7 | Execute | If set, an instruction fetch from the address matching the address (all 32 bits) will halt the processor before the instruction is executed. The PC remains at the address of the instruction that caused the break. |
| 3, 7, 11, 15, 19, 23, 27, 31 | Not used | | |

The BRKE register is one of a number of special registers that are only accessible to programs executing in one of the privileged execution modes. This prevents non-privileged programs from monitoring or modifying these registers and thereby monitoring or modifying their own debug or test conditions.

Providing privileged modes access to the breakpoint registers allows Supervisor Mode debuggers to be implemented for debugging User Mode tasks without needing to stop the processor. This can allow, for example, the main operating system to remain running when debugging User Mode applications.

**Breakpoint Control Registers**

The breakpoint registers can be written and read by privileged modes using special instructions. These instructions can only be executed by programs executing in one of the privileged execution modes. This prevents non-privileged programs from monitoring or modifying their own debug or test conditions.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| %brk0 | 0000 | BRK0 | Breakpoint 0 |
| %brk1 | 0001 | BRK1 | Breakpoint 1 |
| %brk2 | 0010 | BRK2 | Breakpoint 2 |
| %brk3 | 0011 | BRK3 | Breakpoint 3 |
| %brk4 | 0100 | BRK4 | Breakpoint 4 |
| %brk5 | 0101 | BRK5 | Breakpoint 5 |
| %brk6 | 0110 | BRK6 | Breakpoint 6 |
| %brk7 | 0111 | BRK7 | Breakpoint 7 |
| %brk4count | 1000 | BRK4COUNT | BRK4 counter |
| %brk5count | 1001 | BRK5COUNT | BRK5 counter |
| %brk6count | 1010 | BRK6COUNT | BRK6 counter |
| %brk7count | 1011 | BRK7COUNT | BRK7 counter |
| %brk6mask | 1100 | BRK6MASK | BRK6 data mask |
| %brk7mask | 1101 | BRK7MASK | BRK7 data mask |
| %brk6data | 1110 | BRK6DATA | BRK6 data match |
| %brk7data | 1111 | BRK7DATA | BRK7 data match |

**Breakpoint Control Function**

Each breakpoint corresponds to logic gates which implement the control function for the breakpoints. We describe the logic function for breakpoint 7; the other breakpoints are described in a similar fashion with simplifications for breakpoints BRK0 through BRK5 which do not have data predicates, and BRK0 through BRK3 which also do not have count predicates.

The breakpoint BRK7 is triggered if the following four conditions are satisfied:

a) the value on the address bus is exactly the same as the value in the BRK7 register;
b) the mode (read/write/execute) of the memory access cycle is as specified in the corresponding bits in the BRKE register;
c) the value of the BRK7COUNT register is zero;

d) for each bit of the data bus, the corresponding bit in the BRK7MASK register is either 0, or the bit of the data bus matches the corresponding bit in the BRK7DATA register.

If conditions (a) and (b) are met but condition (c) is not met (i.e. the value of BRK7COUNT is not zero) then the value stored in BRK7COUNT is decremented by one.
The behaviour of breakpoint BRK6 is exactly the same as for breakpoint BRK7. The behaviour of breakpoints BRK4 and BRK5 is as above but without condition (d). The behaviour of breakpoints BRK0 through BRK3 is as above but without conditions (c) and (d).

**Debug Interface**

There are two interfaces to the debug registers: via privileged software running on processor and via an external debug system through a "SIF" hardware debug interface. The operation of a SIF interface in particular embodiments is described in more detail in United Kingdom Patent 2 294 137 B, the contents of which are herein incorporated by reference.

Both interfaces can be operated in parallel. This allows, for instance, debug software operating on the processor core to debug an application, with an external debug system monitoring the status of breakpoint counters in real time.

Internal access to the debug registers is via privileged instructions. Such instructions can only be executed by software executing in Supervisor or Interrupt or NMI modes. In non-privileged modes the processor will not execute these instructions.

The invention extends to methods and/or apparatus substantially as herein described with reference to the accompanying drawings.

Any feature in one aspect of the invention may be applied to other aspects of the invention, in any appropriate combination. In particular, method aspects may be applied to apparatus aspects, and vice versa.

The invention also provides a computer program and a computer program product for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein, and a computer readable medium having stored thereon a program for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein.

The invention also provides a signal embodying a computer program for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein, a method of transmitting such a signal, and a computer product having an operating system which supports a computer program for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein.

Furthermore, features implemented in hardware may generally be implemented in software, and vice versa. Any reference to software and hardware features herein should be construed accordingly.

Preferred features of the invention will now be described in the following:- SIF (Serial Interface) Definition document (that document being labelled C4862-TM-005 v3.0, and having 74 pages, the pages being numbered from page 1 of 74 to page 74 of 74); document entitled XAP3 Specification (that document being labelled C7245-S-001 v3.0, and having 151 pages, the pages being numbered from page 1 of 151 to page 151 of 151); document entitled xSIF API (that document being labelled C4862-TM-016 v0.10 and having 43 pages, the pages being numbered from page 1 of 43 to page 43 of 43); document entitled XAP3 Programmer's Manual (that document being labelled C7245-UM-002 v1.0 and having 291 pages, the pages being numbered from 1 of 291 to 291 of 291); and document entitled Summary of Inventions (that document

being labelled C7245-TM-004 v0.7 and having 5 pages, the pages being numbered from page 3 of 10 to page 7 of 10).

It will be understood that the present invention is described below purely by way of example, and modification of detail can be made within the scope of the invention.

Each feature disclosed in the description, and (where appropriate) the drawings may be provided independently or in any appropriate combination.

## Second Aspect: Processor and Interface (XAP4)

The present invention relates to a processor, to an interface associated with the processor, and to methods of operation of the processor and the interface. The invention also relates to an instruction set and to methods of programming operation of the processor and the interface. The invention also relates to a data processing apparatus.

Particular aspects of the invention, and particular embodiments, are now described in sections 1 to 44.

More details of these embodiments are then provided in the following further documents which are included herein:-

- document entitled Summary of Inventions (that document being labelled C7245-TM-004 v0.7 and having 5 pages, the pages being numbered from page 3 of 10 to page 7 of 10)
- document entitled XAP4 Specification (that document being labelled C7432-S-001 v2.0, and having 131 pages, the pages being numbered from page 1 of 131 to page 131 of 131)
- document entitled XAP4 Programmer's Manual (that document being labelled C7432-UM-002 v1.3 and having 224 pages, the pages being numbered from 1 of 224 to 224 of 224)
- document entitled XAP4a Hardware Reference Manual (that document being labelled C7432-UM-011 v1.4 and having 47 pages, the pages being numbered from 1 of 47 to 47 of 47)
- document entitled XAP4a Instruction Set Quick Reference Card (that document being labelled C7432-UM-005 v1.11 and having 4 pages, the pages being numbered from 1 of 4 to 4 of 4)

- document entitled the SIF (Serial Interface) Definition (that document being labelled C4862-TM-005 v3.0, and having 74 pages, the pages being numbered from page 1 of 74 to page 74 of 74)
- document entitled xSIF API (that document being labelled C4862-TM-016 v0.10 and having 43 pages, the pages being numbered from page 1 of 43 to page 43 of 43

The following text references diagrams and text in the above documents, particularly the XAP4 Specification, XAP4a Hardware Reference Manual and SIF Definition documents.

Features disclosed in any of sections 1 to 44 or in any of the further sections may be combined in any appropriate combination.

A brief overview of aspects of the invention, and preferred embodiments, is first provided.

Sections 21 to 44 relate in particular to a processor, for instance a XAP4 processor in the preferred embodiment.

Sections 1 to 20 relate in particular to an interface system, for instance an (SIF) interface system in the preferred embodiment.

Aspects of the SIF and associated processor are described in WO 96/09583, WO 03/048978, GB 2382889, United Kingdom Patent Application Nos. 9419246.5 and 0129144.2, each in the name of Cambridge Consultants Limited and each of which is hereby incorporated by reference.

In one aspect there is provided a processor interface apparatus, comprising at least one processor, an interface connected to the processor and comprising means for writing data to and/or reading data from the processor, control means for controlling operation of the interface, the control means comprising

master and slave modules, and wherein the master controlling module is located on the same chip as the or each processor.

The provision of a master on the same chip as the or each processor (i.e. an on-chip master) provides increased flexibility. In particular, it enables the interface to be used after the design of the chip has been finalised, i.e. in the final application, and not only during the design and debugging of the chip.

Preferably, the apparatus further comprises a master controlling module which is not located on the same chip as the or each processor, i.e. an off-chip master. The off-chip master may be in the form of, or implemented in, a separate piece of hardware, for example, a PC.

In this context, the term "chip" (and hence the terms "on-chip" and "off-chip") preferably refers to any single semiconductor device, such as an ASIC or FPGA including various processors and/or memory, manufactured from silicon, germanium, gallium-arsenide or any other semiconductor material.

The interface is also referred to herein as a SIF interface, which comprises at least one SIF master and at least one SIF slave.

Preferably, the SIF master controls the SIF slave. More preferably, the SIF slave is connected to the or each of the (on-chip) processors.

Thus, the SIF slave enables a SIF master to access one or more on-chip processors (as shown in the diagram in Section 9.2 of the XAP4 Specification).

Preferably, the off-chip SIF master communicates via an external (xSIF) interface.

Preferably, the SIF master can also be on-chip and communicate via an internal (iSIF) interface.

Preferably, read and/or write access to the processors via the (SIF) interface is non-invasive. Thus, the processor's functionality and timing remain unchanged.

Preferably, the on-chip processors can be of different types. Preferably, the processors are XAP processors.

The SIF interface is used in relation to a number of processors, including the XAP4.

The SIF interface provides a mechanism for masters to access the registers, memory and debug instructions of one or more on-chip processors. In the case where the master is an off-chip device the SIF interface is accessed via an external (xSIF) interface. In the case where the master is on-chip the SIF interface is accessed via an internal (iSIF) interface. Preferably, external xSIF interface comprises a serial interface. Preferably, the internal iSIF interface comprises a parallel interface.

In xSIF communications, there is preferably one SIF slave per chip. In a preferred embodiment, a SIF Computer (xSIF master) transfers data to and from one or more on-chip processors via one or more SIF Pods and via one or more SIF slaves. Communications between the SIF pod and the SIF slave use the xSIF interface.

In iSIF communications, in a preferred embodiment, an iSIF master transfers data to and from one or more processors via the SIF slave using the on-chip iSIF (parallel) interface. The iSIF master, SIF slave and processors are normally on the same chip.

The iSIF master may be another processor or a hardware state machine.

SIF technology includes elements of the SIF inside the integrated circuit or chip and external SIF elements found in the SIF Computer and SIF Pods (see

for instance the SIF Definition document, sections 2.1, 2.2 and 11 for an overview of aspects of preferred embodiments).

The XAP4 and SIF Slave module are normally implemented together in the same integrated circuit (XAP4 Specification document, 9.1). The integrated circuit may also contain other circuit elements. The integrated circuit may contain more than one processor, but normally contains just one SIF Slave module.

The XAP4 is a 16-bit processor that may be implemented in an integrated circuit. The integrated circuit may be a standard component or an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA).

In embodiments of the XAP4 processor, the same width is used for data, registers and address and instruction, in accordance with good processor design, and preferably the same memory can be used to store data and store programs and store pointers.

The XAP4 processor is described in detail in the XAP4 Specification document. The Programmer's model for the XAP4 is described in section 5 of the XAP4 Specification. The Instruction Set is described in section 8. The specific SIF Slave module used in the XAP4 is described in section 9.

A preferred embodiment of a SIF system (as shown in the diagrams in sections 2.1 and 2.2 and 11 of the SIF Definition document) comprises three main components: one or more ASICs, a SIF Pod and a SIF computer. Each ASIC comprises a SIF slave module, one or more processors and optionally an iSIF master. The SIF pod and SIF computer are located external to the ASIC. The SIF computer includes a SIF driver, preferably implemented in software. The SIF Pod and the SIF computer together may be referred to as an xSIF master. Alternatively the SIF Pod alone is sometimes referred to as the xSIF master. The SIF computer may be a laptop or desktop computer and may be, for instance, a PC or Mac. All three components are required for

xSIF communication. The SIF Pod and SIF Computer are not required for iSIF communication.

An overview of the architecture of preferred embodiments of the SIF interface system can be seen in the figure shown in sections 2.2 and 11.2 of the SIF Definition document and in sections 9.1 and 9.2 of the XAP4 Specification document.

The integrated circuit is referred to at various places herein as the SIF Slave ASIC (even though it may be an FPGA or standard component).

In the preferred embodiment, the SIF Slave ASIC contains a SIF Slave module, one or more processors, application-specific digital circuitry and application-specific analogue circuitry. In the preferred embodiment the processor is a XAP4 processor. In other embodiments the processor may be, for instance, a XAP1, XAP2, XAP3 or XAP5 processor.

A plurality of SIF Slave ASICs (or FPGAs) may be connected to a single SIF Pod, and a plurality of SIF Pods may be connected to a single SIF Computer, as can be seen for instance in the figure shown in section 2.1 of the SIF Definition document.

The SIF Slave module in the preferred embodiment comprises the external xSIF interface, the internal iSIF interface and control logic.

The xSIF interface comprises a shift register with three interface signals (SIF_CLK, SIF_MOSI, SIF_MISO) and handshake circuitry with one interface signal (SIF_LOADB). See section 7 of the SIF Definition document. The shift register has two inputs; SIF_CLK (to which a clock signal is applied to effect the movement of binary data between subsequent stages of the shift register) and SIF_MOSI (Master-Out, Slave-In) for receiving binary information in a serial manner from an external source (thereby allowing the information to be entered into the shift register and debug system). The shift register has a single output; SIF_MISO (Master-In, Slave-Out) for allowing binary data to be

output in a serial manner from the shift register to an external source. In preferred examples, the shift register is a 64-bit shift register (XAP2) , an 88 bit shift register (XAP3), a 52-bit shift register (XAP4), or a 68-bit shift register (XAP5). Typically, the SIF_CLK, SIF_MOSI, SIF_MISO and SIF_LOADB signals connect the SIF Slave module to the SIF Pod.

The iSIF interface comprises iSIF fields and iSIF control signals. The input fields to iSIF are: address, control and data_write. The output fields from iSIF are: data_read and status. The input control signals to iSIF are: isif_load and isif_cancel. The output control signal from iSIF is isif_done. There are multiplexers for the three iSIF input fields.

The SIF Pod may comprise a hardware-based interface between the SIF Slave ASIC and a personal computer. The SIF Pod may be connected to the SIF Slave ASIC by a 16-wire interface (see SIF Definition document, section 3) and may be connected to the PC via a standard communications interface such as Ethernet, RS232, LPT or USB (Universal Serial Bus). The 16 wire interface includes the xSIF interface.

The SIF Driver in the SIF Computer (which is an xSIF master) may comprise a computer program that translates information between the standard communications interface (such as Ethernet, RS232, LPT or USB) and the xSIF API (Application Programmer's Interface). This enables application programs on the PC to simply control and transfer information with the SIF Slave ASICs. Further computer programs may also be executed on the PC, which programs may utilise the xSIF API to allow data to be written to and read from the SIF Slave devices. This data may then be analysed to determine the operation of the processor, such as for allowing debugging of programs being executed by the processor or processors in the SIF Slave ASIC. Alternatively or additionally, data read from the SIF Slave ASIC may be data relating to the operation of a sensor or other device and the SIF Slave ASIC may then operate as a data logging device.

The SIF interface can be operated on the SIF Slave ASIC concurrently with other software being executed by the processors, such as operating systems and user-mode programs. Concurrent operation may be achieved by time-slicing, which provides time windows during which the ASIC processor performs NOP (No Operation) leaving the SIF Slave module free to execute debug control of the processor, or to access the processor's memory and registers. This mechanism is used both for iSIF and xSIF accesses.

Preferably, xSIF and iSIF use substantially the same instruction set. Thus, the inclusion of iSIF functionality adds very little hardware to the existing xSIF system.

Typically each processor comprises a Core, an MMU (Memory Management Unit), an IVC (Interrupt Vector Controller) and a SIF Slave Module. See the diagrams in the XAP4 Specification document (section 4, p 21) and the XAP4a Hardware Reference Manual (section 2). The Core communicates with memory external to the processor via the Memory Management Unit. In a preferred example, the Memory Management Unit can communicate with external memory including Flash memory (typically used for the storage of programs), RAM (Random Access Memory, typically used for the temporary storage of data variables to be manipulated by programs) and one or more registers.

The memory management unit typically has address and read-write lines and data lines. These signals are typically uni-directional when on-chip. The data lines are typically bi-directional when off-chip.

The SIF Slave Module in the preferred embodiment is connected via each processor to each respective processor's Memory Management Unit, such that the SIF Slave module is able to control the memory control and address signals, thereby enabling it to read and write data from/to the processor's memory. This is used identically for xSIF and iSIF accesses.

In the preferred embodiment, a number of further (control) lines are provided to enable interaction of the processor Core with the SIF Slave module.

In the preferred embodiment for an xSIF interface a number of lines are provided between the SIF Slave module and the SIF Pod, including the SIF_CLK, SIF_MOSI, SIF_MISO and SIF_LOADB lines referred to above. A further input to the SIF Slave ASIC is the chip select line SIF_CS. The 16-wire interface from the SIF Pod contains 4 lines that may be used as separate SIF_CS lines to up to 4 SIF Slave ASICs.

Preferably, the apparatus further comprises at least one control line, and wherein operation of the interface controller is dependent upon a signal applied to the control line.

Preferably, the apparatus processes instructions passed via the internal (iSIF) and external interfaces (xSIF) in alternate fashion.

Preferably, the apparatus further comprises means for arbitrating the processing of instructions passed via the internal (iSIF) and external (xSIF) interfaces.

Preferably, the instructions passed via the external (xSIF) interface take precedence over instructions passed via the internal (iSIF) interface.

According to an aspect of the invention, there is provided a data processing apparatus including a processor, and an interface capable of communicating with the processor, each processor having associated memory, registers and debug instructions, and wherein the interface provides access to the memory, registers and debug instructions of the or each processor.

Preferably, the (SIF) interface comprises at least one (SIF) master module and at least one (SIF) slave module.

Preferably, the SIF slave enables a SIF master to access one or more on-chip processors.

Preferably, the SIF master is provided on-chip. More preferably, the SIF master communicates via an internal (iSIF) interface.

Preferably, the SIF master further comprises an off-chip device. More preferably, the off-chip device communicates via an external (xSIF) interface.

Preferably, access via the SIF interface to the processors is non-invasive. Thus, the processor's functionality and timing remain unchanged by a SIF access.

Preferably, the data processing apparatus is in the form of a single semiconductor device or chip containing one interface and one or more processors.

Preferably, the term semiconductor device is intended to connote an ASIC or FPGA implemented in silicon or germanium or gallium-arsenide or some other semiconductor material. The semiconductor device is also referred to as a chip.

Preferably interface communications are initiated by a master and responded to by a slave.

Preferably, the interface master may be implemented on or off chip.

Preferably, the master can read and write to the memory and registers of one or more processors.

Preferably, the master can also issue debug instructions to one or more processors.

Preferably, communications between the master and slave(s) may be from within the semiconductor device (internal) or from outside the semiconductor device (external).

Preferably, the internal interface is a parallel interface.

Preferably, the external interface is a serial interface.

Preferably, the internal interface master is implemented in hardware in the form of another processor or dedicated state machine.

Preferably, the internal master can read and write registers and memory of one or more processors. This can be used to download software after system reset, to capture and store processor state before turning processor power off, to restore processor state after turning processor power back on, to read memory mapped IO registers in real-time in a non-invasive manner. The IO register could be an ADC (analogue to digital converter).

Preferably, the internal master can also issue debug instructions to one or more processors. This can be used to start, stop, single step, set breakpoint, run to breakpoint and reset one or more processors.

The provision of an internal master enables all of the abovementioned functionality to be used in the final application and not only during product debug.

Preferably, the slave is adapted to support both an internal and external interface. The internal interface is to an internal master. The external interface is to an external master. More preferably, the internal interface is parallel and the external interface is serial.

It is also possible for internal and external masters to simultaneously read and write registers and memory of one or more processors.

Furthermore, it is possible for internal and external masters to issue debug instructions to one or more processors.

## 1. Asynchronous SIF handshake

The SIF Computer can still communicate with the SIF Slave ASIC at high data rates, even when the SIF Slave ASIC is in low power mode. SIF Slave ASIC will often use a slower internal system clock during low power mode, but the SIF remains unaffected.

The benefits of the asynchronous SIF handshake can be exploited by an xSIF or by an iSIF access.

For iSIF the asynchronous handshake is controlled by the isif_load and isif_done signals.

For xSIF, the SIF Shift phase (on SIF_CLK, SIF_MOSI, SIF_MISO) is still fast. The SIF handshake phase (on SIF_LOADB) automatically slows down to accommodate the slower internal system clock.

Master/pod does not need to be aware of change of mode/clocks/speed in the slave. See SIF Definition document (section 6.6).

In one aspect there is provided a processor interface apparatus, comprising a processor, an interface connected to the processor and comprising means for writing data to and/or reading data from the processor, and control means for controlling operation of the interface.

Preferably the processor is adapted to use a first clock and the control means is adapted to use a second clock. Preferably the interface is adapted to operate such that at least one operation is dependent upon the second clock, and preferably such that the timing of the at least one operation is dependent upon the second clock. Preferably the apparatus is arranged so that a request for the interface to write data to and/or read data from the processor is generated dependent upon the second clock. Preferably the control means is adapted to generate the request.

Preferably the apparatus is arranged so that a read operation for reading data from the processor, or a write operation, for writing data to the processor, is carried out in response to the request, and preferably the apparatus is arranged so that the read operation or the write operation is carried out dependent upon the first clock. Preferably the apparatus is arranged so that the read operation or write operation is commenced dependent upon a time slice process. The time slice process is preferably a time slice process of the processor.

Preferably the apparatus is arranged to carry out the write operation or the read operation independently of the second clock.

Preferably the apparatus comprises means for operating at least part of the interface dependent upon the second clock as well as or instead of the first clock. Preferably the means for operating at least part of the interface dependent upon the second clock is adapted to operate at least part of the interface dependent upon the second clock during at least part of a communication process between the interface and the processor.

Preferably the interface comprises a shift register and preferably comprises a SIF Slave module. Preferably the control means comprises a SIF pod, or master, and/or a control computer. Preferably the processor is a XAP4 processor.

Preferably the interface further comprises means for instructing operation of the processor.

Preferably the processor and the interface are on the same circuit board and/or form part of an ASIC or FPGA. Many of the embodiments described herein relate to an ASIC, but may equally relate to an FPGA.

Features of the preferred embodiment are described, in particular, in sections 2.2, 4.3 to 4.5, and 6.6 of the SIF Definition document.

SIF enables clock speeds to be switched in the slave.

An apparatus is provided which provides asynchronous hand shaking between a master and a slave and which does not change the timing of the application running on, or associated with, the slave.

A memory means associated with, and preferably included in, the processor may be controlled either by the processor or by the interface. The memory means may comprise a memory management unit (MMU).

The MMU may be controlled either by the XAP or by the SIF.

In operation of the processor there may be a time slice process. The position of the time slice within the software program may be set by a processor-sif instruction, or fixed in hardware (like a microcode processor on a fixed loop of, say, 256 instructions, with instruction 0 being a fixed SIF cycle).

Further details concerning communication of instructions from a SIF Pod to a SIF Slave, and communication of data between the SIF Pod and the SIF Slave, and the timing of such communication is provided in sections 4, 6, and 7 of the SIF Definition document .

There is provided a processor-sif instruction and a processor-sleepsif instruction so that a programmer can choose when the SIF instructions (from the SIF Computer and SIF Pod) should be executed.

The processor-sleepsif instruction puts the processor into sleep mode which stops instruction execution until a hardware wakeup input is received. In the case where a processor-sleepsif instruction is used, particularly when there are many sensor tasks, there may be a regular wakeup. The wakeup may occur at the rate at which samples are taken in a data acquisition process, for instance between 1 and 20 kHz.

In the preferred embodiment, communication between the SIF Slave module and the processor may take place, for instance, during a period when the processor is in Sleep mode, or during a processor-sif instruction.

In the preferred embodiment, upon entry into Sleep mode the processor typically stops where it is (a software instruction). In the preferred embodiment, the wakeup is typically a hardware input instructing the processor to re-start.

Software can be organised so that after a wakeup, the processor re-starts and branches to software which vectors to a program location dependent upon the type of wakeup. This is similar to an interrupt process but is simpler.

Typically the processor would get on with its work (for instance data acquisition in one example) when it is awake. That work may take a variable amount of time, as it is often data-dependent. There is then a variable time when the processor is asleep (i.e in sleep mode). In that sleep time, SIF Instructions are carried out. As long as the time taken for the SIF instructions to be carried out is less than the time to the next wake-up then operation is straightforward

SIF activity may take place either in during a processor-sif instruction or during a sleep period (which was initiated by a processor-sleepsif instruction).

For sensor applications, use of sleep and wake up in relation to SIF activity works particularly well. This is because none of the inputs are asynchronous. The whole system timing can be controlled from a single clock source.

For radio systems, or other systems which may not follow a regular pattern of activity, it is necessary to respond to asynchronous inputs. In that case it may be particularly advantageous to organise software in a more interrupt-based manner and to use processor-sif instructions rather than processor-sleepsif instructions to control the when communication between the SIF Slave module and the processor occurs.

## 2. Networked SIF

In a further aspect there is provided the ability to simultaneously monitor and debug a plurality of processors and associated memories. Multiprocessor debugging is provided. Multi-processor support in xIDE (a particular Integrated Development Environment) may be used. Data packets flowing between SIF Slaves (e.g. radio or power-line communications devices) may be monitored.

There can be a single xSIF master accessing multiple ASICs each having one xSIF interface. Furthermore, the xSIF master may also access multiple processors within each ASIC. Alternatively there can be a single iSIF master accessing multiple processors via an iSIF within the same ASIC. In this way, the xSIF and iSIF masters may access each processor's associated memory, registers and debug instructions.

In a further aspect, there is provided a device interface apparatus, comprising a plurality of interfaces each comprising respective means for writing data to and/or reading data from a respective device, and control means for controlling operation of the plurality of interfaces. Each device may be a slave and/or may comprise a processor.

Preferably, the apparatus further comprises a plurality of control means each adapted to control a respective plurality of interfaces.

Preferably each interface further comprises means for instructing operation of a respective device.

Preferably the apparatus further comprises a computer adapted to communicate with, and preferably control, the or each control means. Preferably the SIF computer is adapted to receive data from and/or send data to one or more devices via the or each control means.

Preferably the or each computer is adapted to communicate with a respective plurality of control means using a packet based protocol, for instance TCP/IP or Ethernet.

Preferably the apparatus comprises means for detecting errors in packets sent between the or each computer and at least one of the control means.

Preferably, the or each interface comprises a register, preferably a shift register, and preferably the or each interface comprises a SIF slave module. Preferably the control means comprises a SIF Pod and/or a control computer. Preferably the processor is a XAP4 processor.

Preferably the processor and the interface are on the same circuit board and/or form part of an ASIC or FPGA.

Preferably the interface and/or the control means and/or the computer comprises means for debugging the or each device and/or comprises means for capturing data from the or each device and/or comprises means for analysing data from the or each device.

The figure in section 2.1 of the SIF Definition document shows a plurality of networked SIF slaves linked to a SIF Pod/Master.

Networking of SIF slaves allows a single computer to access many SIF Slaves simultaneously which enables, for instance, de-bugging and/or data acquisition, on a wide scale.

Referring to the figure in section 2.1 of the SIF Definition document, software running on each of the SIF slaves (for instance ASICs or FPGAs) shown can be debugged, because packets can be observed from the computer (shown in purple at the top of the figure) going to each of them.

Multiprocessor de-bug may be operated and/or developed in the xIDE environment.

Debugging of multiprocessors in a single chip may also be provided.

This aspect may have particular application to meshing of networked radio devices, particularly when used for communication of non-voice data which a plurality of separate radio devices are able to identify and communicate with each other. By way of example, meshing may be used to control lights in a warehouse. A particular light may be controllable using a radio transmitter. In the case where the radio transmitter is out of range of the light, the radio signal from the transmitter may be passed via a plurality of radio devices (for instance located on other lights) located between the light and the transmitter. The radio devices work out, usually upon power up, how to communicate with each other and pass messages on to the intended recipient device.

Each radio device has a unique number. Each device may be used as stand-alone (often battery powered) device, passing data to and via other devices via radio links.

Each device may be connected via Ethernet in order to enable multi-processor debugging. Thus the devices (slaves) may communicate by radio links between themselves but may also each have an Ethernet connection to the SIF Computer to enable debugging and other data transfer.

The SIF may, for instance, be used for debugging and also for production test and data acquisition.

The SIF enables the reading of and writing to various specified addresses or registers inside a chip in real time without changing its functionality or timing. It is non-invasive. In general the timing of the chip is not changed.

This is particularly advantageous in, for instance, DSP systems having a fixed frequency plan and enables communications to take place between a SIF Computer and a SIF Slave, without changing the SIF Slave's timing.

A chip may be connected up in its real mode to, for instance, a sensor, to a radio, to a USB link, to a RS232 link, to a LPT link or to a CAN link and at the same time may have another wire (e.g SIF) going in which may not be part of the application, allowing monitoring of operation without changing operation or timing.

In normal mode when a processor is running, SIF executions take place in a time slice manner.

**3. SIF enables processor functions to be split across devices.**

In a further aspect there is provided a processor apparatus comprising a first processor, a second processor and interface means adapted to enable communication between the first processor and the second processor.

Preferably the first processor and the second processor are adapted to co-operate together via the interface means, and preferably the first processor is adapted to perform first aspects of a process and the second processor is adapted to perform second aspects of a process.

Preferably the second processor is programmable, so that the second aspects of the process may be varied. Preferably the second aspects comprise aspects relating to a user interface, and/or pricing aspects, and/or language aspects (i.e functions that may need to be modified during product lifetime). Preferably the first aspects comprise critical functionality of the process. Preferably the first aspects comprise aspects of the process which require regulatory approval. Preferably the first aspects comprise functionality that is likely to remain fixed throughout the product lifetime.

Preferably the first processor is adapted to carry out the first aspects of the process in dependence upon instructions that may be stored in ROM (because they are likely to remain fixed). Preferably the second processor is adapted to carry out the second aspects in dependence upon instructions stored in RAM, or Flash memory, or in some other rewritable memory (because they are unlikely to remain fixed).

Preferably the interface further comprises means for instructing operation of the first processor and/or the second processor.

Preferably the first processor is part of an ASIC or FPGA. Preferably the interface comprises a SIF interface. This may be an xSIF or iSIF interface.

In a two-chip solution the second processor is an xSIF master and on a separate chip from the first processor.

In a one-chip solution the two processors are both on the same chip. In this case the second processor is an iSIF master.

SIF enables silicon partitioning. Two-chip solutions to separate fixed operational features from short-term marketing-driven features. This reduces the risk of affecting operation of operational features when altering marketing-driven features.

There are provided two chips, and preferably alteration of functionality of one chip does not affect functionality of the other chip.

As SIF can be used in real applications, it is easier to do partitioning of functionality between chips. Typically, critical functionality may be provided by one chip and less critical functionality (which may vary for particular products and/or for regulatory or marketing reasons).

A SIF may be used for monitoring and data logging and analysis in, for instance, production, monitoring, and analysis systems

In one embodiment, a SIF is used in a gas meter system. An ASIC is used for front end applications. The ASIC in effect implements all of the critical functionality of the system but different products are made for the German, French, British and American markets due to the different regulatory requirements.

A separate ASIC is not produced for each market. Instead, an external microprocessor is provided which talks to the ASIC over the SIF. The microprocessor can access the ASIC in a way which is not invasive. The design is partitioned so that the critical science functionality cannot be tampered with, as it is locked in the ASIC. Whereas the external microprocessor may vary from one market to another.

In another embodiment, there is separation between features of Bluetooth functionality which have obtained regulatory approval, and features which do not require regulatory approval. A chip is provided which includes features of Bluetooth functionality which require or have obtained regulatory approval. A separate software development environment is provided which can be used (for instance by a customer) to customise features of the product, for instance, buttons and LED's. It is ensured that in customising features, the Bluetooth functionality cannot be broken, or changed in such a way that it would be needed to reapply for radio approvals.

## 4. Four SIF fields (e.g address, control, data, status)

In a further aspect there is provided a processor interface apparatus comprising a register adapted to be connected to a processor and to read data from and/or write data to the processor. Preferably the register comprises a plurality of fields, preferably at least three fields, each field comprising at least one bit.

Preferably the plurality of fields comprises at least one of, and preferably all of:- an address field, a control field, a data field, and a status field. See SIF Definition document (for instance section 2.2, section 4, section 7 for description of aspects of the preferred embodiment.

The sizes of each the respective four SIF fields must be the same for the xSIF and iSIF interfaces.

Preferably the address field comprises an address in the processor from which data is to be read or to which data is to be written, and/or the address field comprises information which identifies an operation, for instance a debug operation, to be performed by the processor.

Preferably the data field comprises data to be written to, or data which has been read from, the processor. Preferably the apparatus is adapted to write the data to the address specified in the address field. Preferably the apparatus is adapted to read the data from the address specified in the address field.

Preferably the control field comprises data relating to control of the processor and/or the register and/or a further component of the processor interface apparatus, and/or relating to control of communication between the register and the processor and/or a further component of the processor interface apparatus.

Preferably the control field comprises data relating to at least one of:- error checking, parity checking, processor identification, type of operation to be carried out (for instance read or write), mode information (for instance debug mode, SIF mode), and data-size information.

Preferably the status field comprises data relating to the status of at least one of the processor, the register, and a further component of the processor interface apparatus, and/or relating to communication between at least one of the processor, the register, and a further component of the processor interface apparatus. Preferably the status field comprises data relating to the status of a process carried out by at least one of the processor, the register, and a further component of the processor interface apparatus. Preferably the status field comprises at least one of an error code a parity code and a timestamp. Preferably the contents of the status field are independent of the address.

An error code may comprise a SIF error code or a processor error code. An error code may be included in the status field.

Preferably the register comprises 52 bits. Preferably the address field comprises 16 bits. Preferably the control field comprises 8 bits. Preferably the data field comprises 16 bits. Preferably the status field comprises 12 bits. Such a configuration of the fields is typically used in the XAP4 processor. Thus, the SIF address field is the same width as the processor address bus and the SIF data field is the same width as the processor data bus. However, the SIF control and status fields are independent of any processor bus widths. Advantageously, the SIF control field is identical in all preferred embodiments of the invention. Thus, a single SIF device may be used to debug a variety of processors on the same chip.

Preferably the further component of the apparatus comprises at least one of a SIF Pod or SIF Master, communicating with a SIF Computer running SIF Driver software.

Preferably, the register comprises a shift register, and preferably the register forms part of a SIF Slave module. Preferably the SIF Slave module is adapted to communicate with a SIF Pod or SIF Master. Preferably the processor is a XAP4 processor.

Preferably the processor and the register are on the same circuit board and/or form part of an ASIC.

Four SIF fields may be provided. Each of the four fields may be up to 32 bits long. The four fields may be address, control, data and status fields.

The address and data fields for certain embodiments must be 32 bits in length.

Aspects of the preferred embodiment are described, for example, in section 9 of the XAP4 Specification document, and in particular section 9.1 and section 9.4. More aspects of the preferred embodiment are described in section 7.4 of the SIF Definition document.

Address, control, data, and status fields are described in relation to a SIF Slave for one embodiment in section 7.4 of the SIF Definition document.

Section 9.4.5 of the XAP4 Specification document provides an explanation of the nature of each bit in the four SIF fields in the preferred embodiment.

Transfer of data to and from the address, control, data and status fields in the preferred embodiment is illustrated diagrammatically in the SIF ASIC Block Diagram in section 9.1 of the XAP4 Specification document.

An Application Programming Interface (API) is preferably provided.

The API enables a programmer to program an application which talk to SIF Slaves via SIF Pods. The application may, for instance, be written in xIDE or SIF Toolkit or Matlab. The API may be an xSIF API or a SIF API.

The SIF API has the concept of two fields (address, data), each of which can be up to 32 bits, the lengths being parameterisable. This can be used, for instance, with XAP1 and XAP2 processors. XAP1 may have a 36 bit shift register. XAP2 may have a 64 shift register.

The XAP4 processor requires a 52 bit shift register. The SIF API, can support a shift register having two fields, each up to 32 bit in length (i.e. a total of 64 bits in length), However, the XAP4 processor requires a shift register having four fields.

The xSIF API has the concept of four fields (address, control, data, status), each of which can be up to 32 bits, the lengths being parameterisable. The xSIF API is thus capable of supporting the 52-bit shift register used by the XAP4 processor.

It is preferable for the SIF and xSIF APIs to be portable across several different computer platforms and programming languages.

In alternative embodiments, two fields are provided (address & data), each of 64 bits. This can also support the 52-bit shift register used by the XAP4 processor.

In the preferred embodiment, address and data fields have to be 32 bits, and address is interpreted the same way in the SIF as in the processor.

There is provided multi-processor de-bug within a single chip whilst still only having a single SIF. The SIF may have, for instance, four or five pins.

In the preferred embodiment, the XAP4 SIF allows access to up to 8 processors on the same chip.

Preferably in the control field, there is a write bit which identifies whether a SIF instruction is a read or a write.

Preferably in the control field, there is a debug bit which identifies whether a SIF instruction is a normal instruction or a debug instruction.

The same approach may be used for XAP 2.

Preferably in the control field, there are two size bits which say whether a read or write is to use 8,16 or 32-bit data.

**5. Single Chip (single SIF) support for multiple processors.**

Reference is made to section 13 below, entitled Multi Processor SIF support

## 6. Direct SIF instructions for Reset.

In a further aspect there is provided a processor interface apparatus, comprising a processor, an interface and means for a resetting operation of at least part of the apparatus. Aspects of the preferred embodiment are described in the XAP4 Specification (for instance, section 9.7).

The SIF instruction set is identical for xSIF and iSIF accesses. As such, the SIF reset instructions are identically available to xSIF and iSIF.

Preferably the resetting means is adapted to reset at least part of the processor, at least part of the interface, at least part of at least one further processor, and/or at least part of an associated device. Each of these reset operations can be selected individually or together as a simultaneous group.

Preferably the resetting means is adapted to reset operation of the at least part of the apparatus, preferably the processor or the at least one further processor, independent of the state of operation of the processor or of the at least one further processor.

Alternatively, the resetting means is adapted to reset operation of the at least part of the apparatus, preferably the processor or the at least one further processor, only if the processor or the at least one further processor is in a specified mode, preferably in a debug mode.

Preferably the interface further comprises means for instructing operation of the processor.

Preferably the interface comprises a SIF interface comprising at least one of a SIF Slave module and a SIF Pod. Preferably the processor comprises at least one of a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the processor forms part of an ASIC or FPGA.

In a further aspect there is provided the following features, alone or in any appropriate combination:- direct SIF debug instruction for reset; RST normal/running; RST debug mode; debug/stopped; options everything off chip; ASIC/SIF selected processor; selected user gates.

Description relating to SIF instructions in the preferred embodiment is provided, in particular, section 9 of the XAP4 Specification document, with particular reference to sections 9.5 and 9.7.

Description of direct SIF debug instructions in the preferred embodiment are provided in section 9.5 of the XAP4 Specification document, and in particular in section 9.5.2 which provides a table of XAP4 SIF instructions. Normal SIF instructions are shown first in the table, followed by direct debug instructions, followed by other debug instructions.

It can be seen from section 9.5.1 of the XAP4 Specification document that in preferred embodiments, SIF instructions are either normal SIF instructions or debug SIF instructions. Debug SIF instructions may be direct debug SIF instructions which are executed in the SIF and do not access any processor, or may be processor debug SIF instructions which refer to a selected processor.

The various reset instructions coming from the SIF may be direct SIF instructions.

Two different styles of reset may be provided, which can differentiate between going into the normal reset mode and going into the reset debug mode. Resets may be to everything, or to a chip, or to specific processors, or to the SIF.
Reset instructions from the SIF give precise control of which bits are reset.

How things are put into and come out of reset is important as a de-bug feature.

## 7. SIF status counters and Direct SIF instructions to read them

In a further aspect there is provided a processor interface apparatus, comprising a processor and an interface, the interface comprising means for reading data from and/or writing data to the processor, and the apparatus further comprising means for monitoring operation of the interface and/or operation of the processor and/or communication between the processor and the interface.

Preferably the interface further comprises means for instructing operation of the processor.

Preferably the monitoring means is adapted to monitor at least one of:- the number and/or type and/or timing of data read or write operations between the interface and the processor; the number and/or type and/or timing of errors occurring in the interface, the processor or in communication between the processor and the interface; the number and/or type and/or timing of reset operations and/or cancellation operations occurring in the interface, the processor, or in communication between the interface and the processor; the quantity of data read, write or transferred; and the timing of data readings by the processor.

Preferably the interface comprises a SIF interface comprising at least one of a SIF Slave module and a SIF Pod. Preferably the processor comprises at least one of a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the processor forms part of an ASIC or FPGA.

In a further aspect there is provided at least one self monitoring SIF counter. The SIF itself may monitor the number of reads, writes, errors and number of cancellations, preferably using direct SIF instructions.

Description of aspects of the preferred embodiment is provided in section 9, in particular sections 9.5.2 and 9.6.3 of the XAP4 Specification document.

The SIF itself has at least one counter, and preferably four counters, which monitor SIF activity. Preferably the SIF counters monitor at least one of:- the number of SIF reads since reset, the number of writes, the number of errors which have occurred and the numbers of times SIF operation has been cancelled.

This aspect is particularly useful in connection with the running of, say, a data acquisition system, particularly an overnight data acquisition system, gathering large quantities of data. It enables a user to identify if there had been any problems during the data acquisition process.

In the preferred embodiment, there are Direct SIF Instructions in the SIF Slave module to enable a SIF Computer to read the value of these 4 counters via a SIF Pod. Direct SIF instructions are described in more detail in section 9.5.2 of the XAP4 Specification document.

## 8. SIF cancel methods

In a further aspect there is provided a processor interface apparatus comprising an interface and a processor, the interface being adapted to communicate with the processor when the processor is in a communication mode.

Preferably the processor is adapted to enter the communication mode in response to a signal from the interface. Preferably the processor is adapted to exit the communication mode in response to an indication that a communication process has been completed successfully.

Preferably the interface is adapted to communicate with the processor only when the processor is in a communication mode and the interface is in a communication mode. Preferably the interface is adapted to enter the communication mode in response to a signal from the processor that the processor is in the communication mode.

Preferably the interface and the processor are adapted to communicate using a handshake procedure.

Preferably there is provided means for making the processor and/or the interface exit the communication mode after a specified time delay, preferably a specified time delay from entry into the communication mode. Preferably the means for making the processor and/or the interface exit the communication mode is included in a control means.

Preferably the processor is adapted to exit the communication mode in response to a signal, preferably a signal from the interface, regardless of whether the communication process has been completed successfully or not.

Preferably the processor is adapted to exit the communication mode in response to the interface stopping requesting communication. Preferably the

interface is adapted to stop requesting communication in response to a cancel signal from a or the control means.

The means for cancelling a SIF access is available to both the xSIF and iSIF interfaces.

If an iSIF master has to wait to long for an iSIF access it performs a time-out and cancels the iSIF access with the isif_cancel signal.

If an xSIF master has to wait to long for an xSIF access it performs a time-out and cancels the xSIF access using one of the xSIF cancel methods.

The following paragraphs in this section refer in particular to methods an xSIF master may use to issue a SIF cancel.

Preferably the interface comprises a SIF interface comprising at least one of a SIF Slave module and a SIF Pod. Preferably the processor comprises a XAP4 processor. Preferably the processor forms part of an ASIC or FPGA. Preferably the control means comprises a SIF pod and/or a SIF computer.

In a further aspect there is provided at least one SIF cancel method.

Particular features of the preferred embodiment are described at section 4.7 and section 7, of the SIF Definition document, and section 9 of the XAP4 Specification document.

Preferably the master/pod detects SIF lockup with a time out.

The slave may tell the master that it made an illegal request

In known handshake procedures, a master may queue up a request and a slave may be waiting to process the request but may never get an answer and thus may stay stuck low. In that case, a SIF may not be usable thereafter,

and may produce an error message such as 'SIF error' (even though the error would not have been with the SIF).

Furthermore, if a number of devices are on the same pod, then if one of them pulls SIF_LOADB low then not only is that device stuck but the pod cannot contact any of the devices because the SIF is being shared by the devices.

Examples of when SIF_LOADB may get stuck low, in known handshake procedures, occur when an invalid request is made, for instance when:- an undefined SIF instruction is given; if it is requested to write to a register which is read-only; a request is made which requires being in debug mode but the SIF is not in debug mode; or a request is made whilst a processor is running which requires the processor to be stopped or to execute a processor-sif or processor-sleepsif instruction.

The SIF Cancel Method features provide solutions to the problems outlined in the preceding three paragraphs.

Two methods for cancelling SIF operation may be provided.

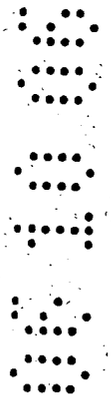
In the first method, the master pulls the slave chip select pin (SIF_CS) low, causing the slave to no longer be selected and clearing any pending SIF cycles which are stuck. Further details of an embodiment of the first method are provided in section 4.7.1 of the SIF Definition document. This method can only cancel an xSIF access.

In the second method, the slave automatically stops pulling the relevant pin low, regardless of actions that have taken place, after a certain number of clock cycles, for instance 32 cycles on SIF_CLK. Thus the SIF is cleared simultaneously for all the slaves. Further details of an embodiment of the first method are provided in section 4.7.2 of the SIF Definition document. This method can cannel an xSIF, an iSIF or both accesses. These different types of cancels are distinguished by the SIF slave sampling SIF_MOSI for the last 8 cycles of SIF_CLK.

The first method typically requires that each slave be cleared in turn. However, usually no more than one slave should be selected at any one time, so no more than one slave should be stuck at any one time in any event. The second method typically clears all slaves simultaneously.

Typically if a pod makes a request that can't be completed by the slave because it has not received the relevant processor-sif or processor-sleepsif instructions it leaves SIF_LOADB stuck low. That tells the pod that it has requested something that it shouldn't have done. Typically the SIF Computer detects a timeout and then issues a SIF CANCEL procedure which will clear the stuck SIF and allow SIF_LOADB to go high again.

Further details concerning lock-up and subsequent cancel procedures in the preferred embodiment are provided, for instance, in section 9.3.2.4 of the XAP4 Specification document.



## 9. SIF timestamp. Use of SIF status field

In a further aspect, there is provided a processor interface apparatus comprising a register, preferably a shift register, adapted to be connected to a processor and to read data from and/or write data to the processor.

Preferably the register is adapted to include time stamp data.

Preferably the register comprises a plurality of fields, preferably at least three fields, and more preferably four fields, each field comprising at least one bit, and preferably the time stamp data is included in at least one of the fields.

Preferably the time stamp data corresponds to a portion of data, preferably to a portion of data written to or read from the register. Preferably the time stamp data indicates when the portion of data was written to or read from the register and/or written to or read from the processor. Preferably the portion of data is data from a device, preferably a sensing device, associated with the processor, and the timestamp indicates when the portion of data was recorded by the device.

Preferably the plurality of fields comprises at least one of, and preferably all of:- an address field, a control field, a data field, and a status field.

Preferably the time stamp data is included in the status field. The corresponding portion of data may be included in the data field.

Preferably the register comprises 52 bits. Preferably the address field comprises 16 bits. Preferably the control field comprises 8 bits. Preferably the data field comprises 16 bits. Preferably the status field comprises 12 bits.

The use of this timestamp in the SIF status field can be exploited by xSIF and iSIF accesses.

Preferably the further component of the apparatus comprises at least one of a SIF Pod or SIF Master together with a SIF Computer running SIF Driver software.

Preferably, the register comprises a shift register, and preferably the register forms part of a SIF Slave module. Preferably the SIF Slave module is adapted to communicate with a SIF Pod, or SIF Master. Preferably the processor is a XAP4 processor.

In a further aspect, a SIF time stamp is provided.

Features of the preferred embodiment are described, for instance, with reference to section 9 of the XAP4 Specification document in particular.

It can be seen in section 9.4.4 of the XAP4 Specification document that the status field is 12 bits in the preferred embodiment, and that the top two bits used are as parity bits (PARITY_S, PARITY_D). STAT is a time stamp, discussed in more detail below.

Processors according to the invention, and in particular XAP processors, may be used, for instance, in software debug, production tests or data acquisition. In preferred embodiments, the SIF is used to acquire data from a processor relating to each of those activities.

In the preferred embodiments, for data acquisition, the master continuously requests data from the slave. The slave attaches a time stamp to each piece of data that it sends to the master. The master can analyse the timestamps to see if it has missed any piece of data, or whether it has received two or more pieces of data from the same sampling period at the master, or whether it has received two or more pieces of data with the same timestamp.

Typically the timestamp is synchronised with a data sampling rate at the master.

Preferably in performing software de-bug, a pod controls transfer of data, and the pod does not have to respond to transactions initiated by the slave.

Preferably, the same SIF interface, with the same pins, is used for debug and for data acquisition. In the preferred embodiment, the data-transfers are always initiated by the SIF Pod and Computer, never by the SIF Slave. This means that in the preferred embodiment the SIF Slave never 'throws' data, even during data-acquisition. The SIF Slave in the preferred embodiment always responds to requests from the SIF Pod, whether it is being used for control or data-acquisition. This makes it much easier to create a deterministic data-transfer system. The problem is that during data-acquisition, some data samples may be acquired more than once, and other samples may be missed. The SIF timestamp is a system solution to this problem.

One example of the use of a data acquisition process is the use of an ADC (analogue to digital converter). Preferably the signal being sampled is over-sampled to ensure that no samples are missed. In characterising an ADC it is important not to miss any samples. It is also important to know whether two copies of the same data have been obtained.

A typical software loop for an ASIC processor comprises:- obtaining new data; then executing the software; then issuing a processor-sleepsif instruction to enable the external SIF Pod to obtain data from the ASIC. In the processor-sleepsif window the SIF can get many reads of the same data. The SIF does not know at that time whether it is in cycle n or cycle n +1.

Preferably there is provided a hardware counter which is incremented at some known rate normally correlated to the data acquisition rate within the chip (for instance about 24kHz).

Preferably the status field is arranged to give extra data which is independent of the address. In the case where a lot of reads are being carried out, the data is from the address requested. The status field is independent of the address requested and can be used for any ASIC register or memory at any address.

Preferably the status field is used to provide a time stamp. In the example given above, the time stamp is an indication of sample number. Accordingly, it is possible to determine whether duplicate copies of the same data have been obtained, or whether samples have been missed.

In certain embodiments, over sampling take place, and the data obtained may be post-processed, for instance in Matlab™. Duplicate copies of data may then be discarded. Accordingly one, and only one, copy of the relevant data is obtained by the post processing.

This feature is particularly useful when using PCs in a data acquisition process as PCs can occasionally miss data acquisition for periods of time, typically 2 to 3 millisecond in length. The use of a time stamp enables identification of periods where data is missing.

Thus 16 bit data can be read, together with a time stamp associated with that 16 bit data.

Considering the time stamp in more detail, shift register bit definitions for the preferred embodiment are provided in, for instance, section 9.4.5 of the XAP4 Specification document.

It can be seen that the shift register bits are divided between address, control, data and status fields in the preferred embodiment, described in section 9.4.5, and that there are 16 address bits, 8 control bits, 16 data bits, and 12 status bits.

Transfer of data to and from the address, control, data and status fields in the preferred embodiment is illustrated diagrammatically in the SIF ASIC Block Diagram in section 9.1 of the XAP4 Specification document.

It can be seen that in the preferred embodiment the 8 bits of the control field are made up of three processor bits (control bits 2:0), one debug bit (control

bit 3), two size bits (control bits 5:4), one parity bit (control bit 6), and one write bit (control bit 7).

The bits making up the address, data, and status fields in the preferred embodiment are also described in more detail in section 9.4.5.

It can be seen from the description of shift register bits 24 to 39 (which make up the 32 bit data field) that in the preferred embodiment the data field may be used for 8, 16, or 32 bit read and write operations.

In the preferred embodiment the status field can be used for a variety of purposes. It can be seen that in the example shown in section 9.4.5, two of the status bits are used as parity bits, six of the status bits are used as error bits, and four of the status bits are used for other status purposes, which may include being used as a time stamp.

In other embodiments, bits, in particular status bits, may be used for other purposes or in different combinations. For instance, nine of the twelve status bits could be used as a time stamp, or for other status purposes.

Preferably, status and control fields are broken down into individual bits, each bit having its own function either alone or in combination with other bits.

Time stamp data may be included in the data field, typically when there is no status field.

The value of the status field is preferably independent of the address field.

The time stamp frequency may be chosen to suit a particular purpose, and may be chosen on command of a user.

Preferably the time stamp is dependent on operation of the SIF and/or of the processor. It may be synchronised with a data acquisition update rate or a software loop rate.

Preferably, the time stamp comprises a count of a number of clock pulses.

Data written to the status field may comprise a software register written to by the processor, as well as or instead of time stamp data.

## 10. Aspects of the processor-sif instruction and SIF instructions

The processor-sif (background debug interface) instruction may be all-1's in 32-bit form, and/or all-0's in 16-bit form. This allows, for example, external SIF control (from the SIF Pod) when the internal ASIC processor data bus is stuck at all-1s or all-0s. This may be exploited by xSIF and iSIF accesses.

In a further aspect there is provided a processor adapted to execute a debug instruction automatically upon the occurrence of a fault. Preferably the fault is an unintended or unforeseen mode of operation of a computer program being executed by the processor. Preferably the debug instruction is encoded by a binary word corresponding to logic levels that are likely to be found on a bus of the processor following a fault. Preferably the bus is a program memory data bus. Preferably the debug instruction is encoded by a binary word in which all bits are 1or as a binary word in which all bits are 0.

Preferably the processor is capable of executing one or more instructions encoded as a binary word having a first length and one or more instructions encoded as a binary word having a second length, wherein the first length is greater than the second length. Preferably the processor is adapted to execute a long debug instruction encoded as a word of the first length and a short debug instruction encoded as a word of the second length. Preferably the short debug instruction is encoded by a word equal to the one's complement of the least significant bits of the long debug instruction.

In the preferred embodiment, operation of the SIF Slave module may be initiated when a dedicated processor-sif instruction is executed by the processor Core. The processor-sif instruction may be stored in program memory (typically on a flash memory device) and read into the Core via the Memory Management Unit.

In certain embodiments, when a processor-sif instruction is read by the Core, the Core completes execution of its current instruction and sends a signal to the SIF Slave module by setting the Core's "SIF" output signal to a defined

logic-level. The execution of further instructions by the Core is suspended until a "CONTINUE" signal is received by the Core from the SIF Slave module (the Core "CONTINUE" input signal typically using the same logic level as the "CONTINUE" output from the SIF Slave module). By suspending the execution of further instructions by the Core in this manner, the SIF Slave module and debugger can be permitted to have uninhibited access to registers in the Core and to the rest of the apparatus. This can also allow debug access via the SIF Slave module to be easily synchronised with software being executed on the processor.

In a preferred example of a processor which executes 32-bit instructions, the SIF instruction is encoded as 0x00000000. In another preferred example of a processor which executes 32-bit instructions, the SIF instruction is encoded as 0xFFFFFFFF. In other words, the SIF instruction is denoted by an instruction in which all bits are zero and/or in which all bits are 1. Thus, when a SIF instruction is read from program memory into the Core, all of the 32 lines of the program memory data bus are high or all of the 32 lines of the program memory data bus are low. Such an arrangement of logic levels on the program memory data bus is also likely to occur following a fault (such as may be caused by incorrect operation of a program being executed by the Core, for example) or when no software has yet been loaded. Therefore, by encoding the SIF instruction with a binary word that is likely to be found on the program memory data bus following a fault, the processor can be caused automatically to execute a SIF instruction immediately following such a fault. This can allow operation of the SIF module to be initiated in response to the fault thereby allowing software to be debugged.

In a preferred example in which the processor is capable of executing both 16-bit an 32-bit instructions, in 16-bit instruction mode the SIF instruction is encoded as 0x0000 and in 32-bit mode as 0xFFFFFFFF.

Features of the preferred embodiment are described in more detail with reference to, for instance, sections 8 and 9 of the XAP4 Processor document.

In particular sections 9.2, 9.3, 9.5, and 9.7, of the XAP4 Specification document.

Performance of an operation may require a processor-sif instruction. In certain circumstances, errors may result in the data bus being stuck at all 1s or all 0s.

By making the processor-sif debug instruction all 1s or all 0s, external SIF control, and debugging, is enabled when such errors occur.

Description of direct SIF debug instructions in the preferred embodiment are provided in section 9.5 of the XAP4 Specification document, and in particular in section 9.5.2 which provides a table of XAP4 SIF instructions. Normal SIF instructions are shown first in the table, followed by direct debug instructions, followed by other debug instructions.

As mentioned above, it can be seen from the table in section 9.5.2 of the XAP4 Specification document that in preferred embodiments, SIF instructions are either normal SIF instructions or debug SIF instructions. Debug SIF instructions may be direct debug SIF instructions which are executed in the SIF and do not access any processor, or may be processor debug SIF instructions which refer to a selected processor.

The first two instructions in the table in section 9.5.2 of the XAP4 Specification document, by way of example, are SIF normal instructions, for reading from and writing to memory. The size of the data field, whether 8 bit, or 16 bit, is specified. In order to carry out the read or write process, a processor-sif or a processor-sleepsif instruction is first required from the processor.

Some debug instructions may be executed immediately and may not need to wait for a SIF cycle.

In the preferred embodiment, instructions for, for instance, going into debug mode; coming out of debug mode; reading status; reading version number; and reading licence number, may be carried out without being in debug mode.

Once in debug mode, a wide range of actions may be performed, for instance Stop, Run, Single Step, Run to Break.

This breakdown of when SIF instructions are executed in the preferred embodiment is explained better in more detail in section 9.3 of the XAP4 Specification document.

In the normal mode of the preferred embodiment, a SIF operation comprises three stages:- shift in stage, handshake stage and shift out stage. See XAP4 Specification document, section 9.3.

In certain circumstances there may be some ambiguity, as an operation may be validly waiting for something to happen, may be operating with a time delay, or may be stuck (an error having occurred). In each case there may a significant amount of time after the shift in stage, before a handshake occurs, during which time it may not be clear whether the processor is operating normally, and the handshake will occur in due course, or whether the processor has locked up.

In the preferred embodiment, a SIF lock-up may be cleared with SIF Cancel (described in more detail at, for instance, section 9.3.2.4 of the XAP4 Specification document) as discussed in section 8 above.

**11. 1st SIF clock edge = out, 2nd edge = in**

The following feature applies to serial interface accesses, and hence in general applies to xSIF only. However, in a case where iSIF employs a serial (instead of a parallel) interface this feature may also be employed by iSIF.

In a further aspect there is provided a processor interface apparatus, comprising a processor, an interface associated with the processor and control means.

Preferably the processor comprises a first clock, and preferably timings of at least some operations, and preferably all operations, of the processor are dependent upon the first clock. Preferably the first clock is a continuously running clock. Preferably the first clock is the ASIC system clock.

Preferably the apparatus comprises means for generating an interface clock signal, and preferably timing of at least some operations of the interface are dependent upon the interface clock signal. In the preferred embodiment, the interface clock signal is SIF_CLK.

Preferably the interface clock signal comprises a series of pulses.

Preferably the interface is adapted to carry out a read operation, preferably a parallel read operation, to read data from the processor. Preferably the interface is adapted to carry out a write operation, preferably a parallel write operation, to write data to the processor. Preferably the read operation and the write operation are carried out by the interface in response to a request from the control means.

Preferably the read operation is carried out dependent upon the interface clock signal.

Preferably the interface comprises a shift register.

Preferably the read operation comprises a load operation comprising loading data into the shift register, preferably from the processor.

Preferably the load operation of the read operation occurs in response to the first edge of the first pulse in the series of pulses of the interface clock signal. Preferably the first edge is a rising edge.

Preferably the write operation comprises writing data from the shift register to the processor, later followed by a load operation comprising loading further data to the shift register.

Preferably the load operation of the write operation occurs in response to the first edge of the first pulse in the series of pulses of the interface clock signal. Preferably the first edge is a rising edge.

Preferably the interface clock signal is held low when no read or write operation is being performed, thus providing lower power consumption.

Preferably the shift register comprises address, control, data, and status fields.

Preferably the data or the further data comprises at least one of address field data, control field data, data field data and status field data.

The load operation may be commenced on a rising edge of a pulse, preferably the first pulse in the series of pulses.

Having the first edge as the shift out edge makes it easier to implement the parallel load mechanism into the shift register in the interface. The reason this is easier is that all of the flip flops in the shift register can use a single ungated clock.

Preferably a load operation commences in response to a rising edge of the interface clock signal.

Preferably the interface comprises a SIF. Preferably the control means comprises a SIF Pod or master and/or a control computer. Preferably the processor comprises an 8bit, 16 bit, 32 bit or 64 bit processor. The processor may be a XAP1 processor, a XAP2 processor or a XAP4 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Preferably the ffirst SIF clock edge is used for parallel load and shift out. Preferably the second edge is used to shift in. This makes the load then shift work properly, ie capture to shift reg (=load) on posedge of SIF_CLK (which makes the msb visible on SIF_MISO), then shift in (to SIF_MOSI) on negedge of SIF_CLK.

There is provided a shift register procedure comprising parallel load first, shift second, using an asynchronous design in which a clock is not always running. Different clock domains are provided.

Features of the preferred embodiment are described with reference, for example, to sections 4, 6 and 7, and in particular sections 4.4.1, 6.2 to 6.6, and 7.2 to 7.6, of the SIF Definition document. Particular reference is made to the figures in sections 4.4.1 and 7.6.

The rising and the falling edge are not symmetrical, one of them is a first.

Data may be clocked into and out of shift registers on opposite edges of a clock signal when coming off devices.

In the preferred embodiment there is provided parallel circuitry inside a slave.

A parallel load may be provided in the SIF Module, and the parallel loaded data may then be sent to the SIF pod here. The parallel load in the SIF Module and the shift out may be on the first edge and the shift in may be on the second edge when doing a load and then a shift.

Clocking in may occur on a negative edge of the interface clock and clocking out may occur on a positive edge of the interface clock and the interface clock may be left low normally.

In the preferred embodiment, operations in the shift register (at the SIF, or SIF Module) are clocked by SIF_CLK, whilst operations in the processor and most of the SIF Slave ASIC are clocked by the system clock. SIF_CLK may be left low, which is a low power state, when not required. The system clock is typically a continuously running clock.

In the preferred embodiment the system clock is used for handshaking on SIF_LOADB. The system clock is typically used to update data capture and the status capture registers and, when complete, SIF_LOADB is allowed to go back to a high state. The Master may then shift out data on the shift register on the SIF_MISO line from the Slave (ASIC or FPGA) to the Master (Pod).

In particular embodiments the contents of certain registers may be transferred to the shift register on the first edge of SIF_CLK.

The system clock may be a continuous running clock. SIF_CLK may comprise a finite series of pulses, for instance 32 pulses. SIF_CLK may be generated externally by the SIF Master (Pod) and is an input to the SIF Slave (ASIC or FPGA).

Parallel load from processor to SIF and serial load from SIF to SIF pod (master) may be provided.

Preferably there is a separation in clock domains between processor and SIF Slave module.

## 12. SIF command mode

The SIF command mode enables plug and play. Different SIFs may be of different length. – The SIF command mode may be used mainly for query functions. Use of the SIF command mode enables query functions without the need to add extra pins, without affecting normal operation and without having any ambiguity of command.

SIF command mode is in generally used for xSIF accesses.

In a further aspect there is provided an interface apparatus comprising an interface, adapted to communicate with a device, for instance a processor, and a control means adapted to communicate with and/or control the interface, the interface being adapted to send data concerning itself to the control means and the control means being adapted to configure itself, or the interface, dependent upon the data.

Preferably the interface is adapted to send the data concerning itself to the control means when in a specified mode, preferably a Command mode. Preferably there is provide a command set for use in the Command mode, the command set comprising a plurality of commands. Preferably the interface is one of a plurality of interfaces, and preferably each interface is adapted to recognise a respective sub-set of the command set or all of the command set. Preferably all of the interfaces recognise a common sub-set of the command set.

By providing such features, interface interrogation functionality is provided without the need to add additional hardware, for instance additional pins, and without affecting existing modes of operation. Existing modes of operation may comprise, for instance, Normal SIF instructions, Debug SIF instructions, and SIF Cancel operations.

Preferably the data comprises at least one of a specific processor or chip identifier, or a generic interface identifier identifying the type of interface, preferably the type of SIF interface, or data representing at least one individual characteristic of at least one component of the interface, preferably a SIF interface. The at least one individual characteristic may comprise a field length, for instance an address, control, data, or status field length. Also the at least one individual characteristic may comprise information concerning the bits within a field, in particular within the control and/or status fields.

Preferably, the interface may be one of a plurality of interfaces. Preferably each interface comprises a respective shift register. Preferably each shift register comprises a plurality of fields. Preferably, for each shift register,

Preferably, the interface is adapted to send data concerning itself in response to a request signal from the control means. Preferably the request signal comprises a specified sequence of signals. Preferably the apparatus is arranged so that the request signal may be sent from the control means to the interface over a channel which is normally used to send other signals between the control means and the interface, and the interface is adapted to recognise and respond to the request signal when sent over the channel.

Preferably the request signal comprises a specified sequence of 8, 16, 32, 64, 128, or 256 bits or other data units. Preferably the interface is adapted to enter a command mode in response to the request signal. Preferably the interface is adapted to exit the command mode in response to a further request signal, which may be the same as or different to the request signal.

Preferably the interface comprises a SIF. Preferably the control means comprises a SIF Pod and/or a control computer. Preferably the processor comprises an 8bit, 16 bit, 32 bit or 64 bit processor. The processor may be a XAP1 processor, a XAP2 processor a XAP3 processor or a XAP4 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the

interface are on the same circuit board. Preferably the request signal represents a command in Command mode.

Features of the preferred embodiment are described with reference to, for instance, section 4, and in particular section 4.8, of the SIF Definition document, and section 9, in particular sections 9.2 (in particular 9.2.1.2) and 9.3, of the XAP4 Specification document.

SIF normally operates in Normal mode. In the preferred embodiment a Command mode is provided for interrogation purposes, in particular to ascertain characteristics of a particular SIF.

In preferred embodiments, all SIFs recognise certain common commands in Command mode, and all SIFs enter the Command mode in the same way. However, particular SIFs may not be adapted to recognise all commands in Command mode.

Different SIFs may have shift registers of different lengths.

In preferred embodiments, all SIFs recognise request commands requesting a chip identifier, or SIF type, or number or type of processors associated with the SIF, or SIF field length, or information concerning bits within a SIF field.

In preferred embodiments, in Command mode, if the SIF does not recognise a command it generates a response of 0xFF. The range of responses to any valid command in Command mode is 0x00 to 0xFE.

The Command mode enables a SIF plug and play functionality.

SIF characteristics may vary, and in particular characteristics of the SIF shift register or SIF shift register fields, in particular SIF shift register length or SIF shift register field length, may vary for different SIFs and/or for different processors with which a SIF may be associated.

For instance, for XAP1, XAP2, XAP3, XAP4 or XAP5 processors the SIF shift register length is typically 32 bits, 64 bits, 88 bits and 52 bits respectively.

In the preferred embodiment, a command mode is provided. When the SIF is in the command mode, the master can interrogate the SIF, typically using query functions. When in the command mode, in the preferred embodiment, the SIF does not access the associated processor; the SIF is returned to normal mode when access to the associated processor by the SIF is required.

In the preferred embodiment, the command mode is entered in response to a series of 256 clocks/bits representing successive 'Hex 33333...' signals on the SIF_MOSI line.

In command mode, in the preferred embodiment, all commands are 8 bits in and 8 bits out.

When the SIF is in command mode, the master can ask various questions, for instance 'what is the address field length?' 'what is the control field length?' 'What is the data field length?' 'How many processors do you have?' 'what types of processors are they at the various places?'. There is provided an instruction set which the SIF can recognise in command mode.

Then when you either shift in an instruction of zeros or you pull chip select low or you pull SIF_LOADB low it immediately reverts to normal mode.

Thus the ability to interrogate an interface, or part of an interface, and to set up control means, so as to communicate with and/or control the interface, is provided without adding any extra pins to the SIF and in a way which does not affect normal operation. In the preferred embodiment command mode can be entered without having to pull SIF_LOADB low.

Typically, in the preferred embodiment, when a device is first plugged in, command mode is entered, the device is interrogated, the apparatus is tailored to the device if necessary, and then normal mode is entered.

## 13. Multi Processor SIF support

Multiple processors per SIF - multi processor support, direct SIF instruction (talking direct to the SIF). Multiprocessor support is provided for xSIF and iSIF accesses.

In a further aspect there is provided apparatus comprising an interface adapted to be connected to a plurality of processors, and adapted to communicate with each processor individually. Preferably the interface is also adapted to communicate with the plurality of processors together.

Preferably the interface is responsive to a processor identifier. Preferably each of the processors is responsive to at least one respective processor identifier. Preferably the apparatus is arranged to establish communication between the interface and one or more of the processors in dependence on a processor identifier. Preferably the processor identifier is included in a portion of data, and the interface is adapted to process the portion of data, and in particular to send the portion of data to, or receive the portion of data from, a particular processor or processors, in dependence upon the processor identifier. Preferably the interface is adapted to read data from or write data to a particular processor or processors in dependence upon the device identifier.

Preferably the apparatus is adapted to read the portion of data into and/or out of a register, preferably a shift register.

Preferably the interface is adapted to send a reset instruction to a particular processor or processors identified by the processor identifier. The reset instruction may be one of a plurality of reset instructions. One of the reset instructions may be such as to reset the particular processor or processors to a normal mode of operation, preferably running in the normal mode of operation. Another one of the reset instructions may be such as to reset the particular processor or processors to a debug mode of operation, preferably stopped in the debug mode of operation.

A processor identifier may identify a plurality of the processors, preferably all of the processors.

Preferably the interface comprises a SIF or SIF module, preferably each processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably each processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably each interface forms part of an interface apparatus which preferably includes means for debugging, or for acquiring data from, the interface and/or the or each processor. The processors may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processors and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 4, in particular section 4.8, and section 7, in particular section 7.2, of the SIF Definition document, and section 9 of the XAP4 Specification document.

In the preferred embodiment, particular bits may be allocated in every SIF instruction which identify a processor, if any, with which the instruction is concerned.

Typically one SIF per chip is provided, and multiple processors per SIF. Multiple processors per chip may be provided. The chip may comprise an ASIC or FPGA. Each SIF instruction may be directed to any one or ones of the multiple processors.

In the preferred embodiment, there are various ways which allow a master to talk just to the SIF. There are various ways in which the master can find out about the SIF as well as or instead of about a plurality of processors associated with the SIF. For instance command mode enables information concerning the SIF to be obtained. Direct de-bug SIF instructions in normal mode also enable information concerning the SIF to be obtained.

Processor debug SIF instructions may specify one or more of the plurality of processors to which those instructions are to be sent.

By way of example, a particular chip may be connected to a computer. The computer, using a SIF master installed on or connected to the computer, may go into command mode.

In command mode, the SIF master may talk to the SIF to find out how many processors there are on the chip and what types they are. It may read various addresses and subsequently (when having left normal mode) communicate with any of the processors. It may also find out from the SIF what the lengths are of the various SIF fields. It may also obtain an ASIC identifier, if one exists which would tell it more about the chip it is connected to. Subsequently the computer may put the system into normal mode and could then, for instance, debug processor 0 while leaving processor 1 and 2 running. Processor 0 and 1 may be, say, XAP4 whereas processor 2 may be, say, XAP2. Individual processors can be reset or the whole chip can be reset. Single step or run to break point operation may be set on one processor whilst another is set to run normally. All of these functions are provided in the preferred embodiment without adding any extra pins.

In a further aspect there is provided a processor interface apparatus, comprising an interface for writing data to and/or reading data from a plurality of processors, and control means for controlling operation of the interface. Using a single interface to access several processors uses fewer chip pins than having a separate interface for each processor. For example, a single SIF interface with 4 chip pins may be used to access 8 on-chip processors.

Preferably the interface further comprises means for instructing operation of each of the plurality of processors.

Preferably the plurality of processors form part of a single chip. Preferably the plurality of processors form part of an ASIC or FPGA.

Preferably the interface comprises a SIF Slave module. Preferably the control means comprises a SIF Pod and/or a control computer.

Preferably each of the plurality of processors comprises one of a XAP1 processor, a XAP2 processor and a XAP4 processor.

Single chip support for multiple processors is provided. Aspects of this feature in the preferred embodiment are described, for instance, section 9 of the XAP4 Specification document.

## 14. Pipeline SIF reads

In a further aspect there is provided an interface adapted to write data to a device, for instance a processor, in a writing operation and/or read data from the device in a reading operation, the interface being adapted to perform a series of operations, and the interface being further adapted to carry out part of one operation in the series at the same time as carrying out part of at least one other operation in the series, preferably the next operation in the series. Preferably the interface is adapted to operate in pipeline fashion.

Thus, higher data rate operations are enabled than if the interface is operated in non-pipeline fashion. Speed is increased when pipelining is applied to serial xSIF accesses or parallel iSIF accesses.

Preferably the series of operations comprises a series of writing operations and/or a series of reading operations.

The interface preferably comprises a shift register. Preferably the interface is adapted to shift in a part of a set of data from a reading or writing operation in a series of operations at the same time as shifting out a part of a set of data from another reading or writing operation in the series of operations.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the interface forms part of an interface apparatus which preferably includes means for debugging, or for acquiring data from, the interface and/or the or each processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 4, in particular sections 4.4 and 4.5, of the SIF Definition document, and section 9 of the XAP4 Specification document.

In the preferred embodiment the SIF comprises address, control, data and status fields. The SIF may shift in address and control and shift out data and status simultaneously.

In one example using the preferred embodiment, a data acquisition process may require that, say, 10,000 data reads at 12 addresses are carried out. The SIF would operate in a cycle in that example, shifting out data and status data from one read at the same time as shifting in address and control data for the next in a pipeline manner.

## 15. Continuous Shift Reg from SIF_MOSI to SIF_MISO

The following paragraphs apply to xSIF and not to iSIF.

In a further aspect there is provided an interface apparatus comprising an interface associated with a device, for instance a processor, and a control means adapted to communicate with and/or control the interface, the apparatus being arranged so that data can be sent between the interface and the control means, the apparatus further comprising monitoring means for monitoring changes in data having occurred between sending of the data and receipt of the data.

Preferably the monitoring means is adapted to monitor data received at the interface and/or at the control means. Preferably the interface comprises a shift register, preferably a continuous shift register. Preferably the interface is adapted to suspend communication with the processor during operation of the monitoring means.

There is also provided a method of checking for errors in data transmission or reception in an interface apparatus, preferably an interface apparatus as described herein, comprising sending data between an interface and a control means and monitoring any changes in the data between sending of the data and receipt of the data. Preferably the monitoring step takes place upon receipt of the data at the interface. Alternatively or additionally, the monitoring step may take place upon receipt of the data at the control means.

Preferably the method further comprises sending the data between the control means and at least one further device.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the at least one further device comprises a control computer or a driver

installed on a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 5, in particular section 5.3.2, and section 6, in particular section 6.6, of the SIF Definition document, and in section 9 of the XAP4 Specification document

In practice, in the preferred embodiment, data is generally passed between the processor, the SIF, the SIF master, and a computer connected to the SIF master. Data may become corrupted during its passage between those devices, and in particular may become corrupted during its passage over the wires which, in the preferred embodiment, connect the devices.

Using the continuous shift register in the SIF, data can be passed backwards and forwards between the SIF and other devices and checks can be performed to see if data has been corrupted during its passage between devices.

The use of SIF_LOADB is particularly useful in this context as, in the preferred embodiment, when SIF_LOADB is kept high data can still be written to and read from the SIF shift register by the SIF master without there being any effect on SIF slave (see section 6.6 of the SIF Definition document in particular). Thus data can be passed between the SIF and the SIF master, and between the SIF master and other devices if required, and data integrity checks to be carried out, without any effect on the SIF slave.

Continuous shift register from SIF_MOSI to SIF_MISO is useful for data integrity checking. It enables the SIF computer and SIF Pod to rotate data at different rates and discover the maximum rate without data error. It also

enables the SIF computer to identify a type of SIF slave from the shift register length.

**16. Synchronise FIFO with SIF_LOADB or isif_load for data acquisition**

xSIF accesses may synchronise a FIFO with SIF_LOADB for data acquisition, without extra pins.

iSIF accesses may synchronise a FIFO with isif_load for data acquisition.

In a further aspect there is provided an interface apparatus comprising an interface and a processor, the processor comprising a register for temporary storage of data, and the interface being adapted to read data stored in the register.

Preferably the apparatus further comprises means for sending a signal to the interface indicating that the register has been updated, the interface being adapted to read updated data stored in the register in response to the update signal.

Preferably the apparatus further comprises control means adapted to communicate with the interface. Preferably the apparatus comprises means for sending to the control means data read from the register by the interface.

Preferably the control means is adapted to establish a communication session with the interface. Preferably the sending means is adapted to send the data from the interface to the control means during a communication session. Preferably the sending means is adapted to maintain a communication session between the control means and the interface pending updated data becoming available to be sent from the interface to the control means.

Preferably the interface comprises an interface register arranged so that data read from the register is read to the interface register. Preferably the register comprises a FIFO (first in, first out).

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Preferably when a master issues a SIF instruction to read the FIFO address the SIF_LOADB or isif_load handshake will not be completed until the FIFO has been updated with new data and then the processor executes a processor-sif instruction or a processor-sleepsif instruction.

Features of the preferred embodiment are described with reference to, for instance, section 4, in particular section 4.6, and section 5,of the SIF Definition document.

There is provided a FIFO inside a slave. An associated SIF may read data from the FIFO.

A handshake procedure may be provided between the SIF and an xSIF master. The xSIF master may make a request and the SIF_LOADB may stay low until the new data is obtained from the FIFO. The SIF would then release the new data to the xSIF master.

A handshake procedure may be provided between the SIF and an iSIF master. The iSIF master may make a request with isif_load and the isif_done signal may stay low until the new data is obtained from the FIFO. The SIF would then release the new data to the iSIF master.

## 17. SIF Pod protocol and associated error correction

In a further aspect there is provided apparatus comprising a control means, preferably a SIF pod, adapted to be connected to a control computer, preferably a SIF computer, and to send data to and/or receive data from the control computer, and means for performing an error checking and correction procedure on the data. Preferably the error checking procedure is based upon checking CRC codes included in the data.

Preferably the SIF pod is adapted to be connected to the control computer via an Ethernet or USB or RS232 or LPT link.

Features of the preferred embodiment are described with reference to, for instance, section 2, in particular sections 2.1, 2.2, 2.3 and 2.6, and section 5, in particular 5.1.1 and 5.4, of the SIF Definition document.

## 18. SIF error reporting

SIF error reporting - SIF error codes, processor error codes - from pod to slave and/or slave to pod. SIF error reporting is provided for both xSIF and iSIF accesses.

In a further aspect there is provided an apparatus comprising an interface adapted to communicate with at least one processor, and a control means adapted to communicate with, and/or control operation of, the interface, the apparatus further comprising means for monitoring data received at the or each processor and/or at the interface and/or at the control means, and for detecting an error in the data.

Preferably the monitoring means is adapted to generate an error message upon detection of an error, and preferably is further adapted to send the error message to a source from which the data was sent. The source comprises one of the processor, the interface, or the control means.

The monitoring means is preferably adapted to identify the source of an error, and may be adapted to identify whether the error occurred at the or one of the processors or the interface.

The data may comprise an instruction. The monitoring means may be adapted to detect an error if the processor or the interface or the control means is not in a correct mode to carry out the instruction.

Preferably the data comprises at least one parity bit and the monitoring means is adapted to carry out a parity checking procedure with respect to the or each parity bit, and to detect an error in dependence upon the parity checking procedure.

Preferably the data comprises a plurality of fields, and at least one parity bit is included in or associated with at least one of the plurality of fields. Preferably

the plurality of fields comprises at least one of an address field, a data field, a control field, and a status field.

In the case where the error occurs at the interface, the monitoring means is adapted to identify whether the error is a parity error, an unrecognised Direct SIF instruction, an invalid data value, or an unrecognised processor identifier.

In the case where the error occurs at the processor or one of the processors, the monitoring means is adapted to identify whether the error is an unrecognised Normal SIF instruction, an invalid data value, a SIF write to a read only register, a SIF read from a write only register, processor not in debug mode (when it should be for a particular SIF instruction to be recognised), or processor not stopped (when it should be for a particular SIF instruction to be recognised).

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Provision of such features is particularly important in the case of networks of processors and interfaces, where having effective debug functionality is necessary.

Features of the preferred embodiment are described with reference to, for instance, section 5 of the SIF Definition document and section 9, in particular section 9.6, and in particular 9.6.2, of the XAP4 Specification document.

## 19. SIF parity checking

Parity bit checking, in and/or out of the SIF, may be provided. Parity bit checking is provided for both xSIF and iSIF accesses.

In a further aspect there is provided an apparatus comprising an interface adapted to communicate with a processor, and a control means adapted to communicate with, and/or control operation of, the interface, the apparatus further comprising monitoring means for monitoring communication of data, preferably between the control means and the interface.

Preferably the monitoring means is adapted to perform an error checking procedure to determine whether there has been an error in communicating data between the interface and the control means.

Preferably the data comprises at least one parity bit and the monitoring means is adapted to carry out a parity checking procedure with respect to the or each parity bit.

Preferably the data comprises a plurality of fields, and at least one parity bit is included in or associated with at least one of the plurality of fields. Preferably the plurality of fields comprise at least one of an address field, a data field, a control field, and a status field.

Preferably the control field contains a parity bit, preferably a PARITY_AC bit. Preferably that parity bit is odd parity (odd number of 1s) for the combined address and control fields.

Preferably the status field contains a parity bit, preferably a PARITY_D bit. Preferably that parity bit is odd parity (odd number of 1s) for the combined data field and the parity bit itself.

Preferably the status field contains a further parity bit, preferably a PARITY_S bit. Preferably that parity bit is odd parity (odd number of 1s) for the status field (including the further parity bit itself).

Preferably the apparatus is adapted to carry out a further operation on the data, or on a field included in the data, in dependence on the outcome of the parity checking procedure.

Preferably the monitoring means is adapted to monitor communication of data between the interface and the processor, and preferably the monitoring means is adapted to perform an error checking procedure to determine whether there has been an error in communicating data between the interface and the processor.

Preferably the monitoring means is adapted to monitor data received by the interface from the control means and/or is adapted to monitor data received by the control means from the interface.

Preferably the interface comprises a SIF, preferably the processor comprises an 8-bit, 16-bit, 32-bit, or 64-bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 5, in particular section 5.3.1, and section 7, in particular sections 7.4.2 and 7.4.4, of the SIF Definition document and section 9, in particular section 9.6, in particular section 9.6.1, of the XAP4 Specification document.

## 20. Serial interface parity checking method that exposes stuck-at faults

The following feature applies to serial interface accesses, and hence in general applies to xSIF only. However, in a case where iSIF employs a serial (instead of a parallel) interface this feature may also be employed by iSIF.

In a further aspect there is provided an apparatus comprising means for detecting errors, the error detecting means being adapted to generate at least one parity bit in dependence on the parity of a plurality of subject bits.

Preferably the at least one parity bit is one parity bit.

Preferably the plurality of subject bits includes the or each parity bit.

Preferably the plurality of subject bits is an even number of bits.

In the case where the plurality of subject bits is an even number of bits the error detecting means will generate the at least one parity bit so that the parity of the subject bits and the at least one parity bit together is even (even number of 1s) in the case where the subject bits are all zeros or all ones (which is indicative of an error in preferred embodiments).

In known parity checking systems a single parity bit will only expose a single bit error. In a system according to the present aspect, a word of all 1s or a word of all 0s will also be exposed as an error. This is useful for serial systems as it will expose the serial wire being stuck at 1 and being stuck at 0. In the preferred embodiment, the PARITY_AC bit exposes stuck at faults on SIF_MOSI (see for instance section 9.6.1.1 of the XAP4 Specification document) and the PARITY_S bit exposes stuck at faults on SIF_MISO (see for instance section 9.6.1.3 of the XAP4 Specification document).

Preferably, the error detecting means is adapted to detect errors in a serial operation, preferably a serial read or write operation, in dependence on the at least one parity bit.

Preferably the error detecting means is adapted to indicate a parity error when the subject bits are characteristic of an error having occurred and preferably is adapted to indicate a parity error when the subject bits are all zeros or all ones.

Preferably the error detecting means is adapted to generate the at least one parity bit so that the parity of the subject bits and the at least one parity bit together is even in the case where the subject bits are characteristic of an error having occurred.

Preferably the apparatus is an interface apparatus, comprising an interface adapted to communicate with a processor and/or a control means. Preferably the error detecting means is adapted to detect errors occurring in the sending of the subject bits between the interface and the control means and/or the processor.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the control means comprises a SIF Pod and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 9, and in particular sections 9.4 to 9.6, of the XAP4 Specification document.

A field may have an even or odd number of bits. Parity checking may be even or odd. Accordingly, there are four possible combinations. Having an even number of bits and having odd parity checking may be particularly advantageous. In that case, in error free operation there will be an odd number of 1s in the field, the field having an even number of bits.

By way of example, in the preferred embodiment a line, for instance the SIF_MOSI line or the SIF_MISO line, could be stuck high or low, and that would cause a parity error.

In the preferred embodiment the PARITY_AC bit is dependent on a section 40 bits long and thus having an even number of bits, and the PARITY_S bit is dependent on a section having 16 bits and thus also having an even number of bits.

In the preferred embodiment, the section on which the PARITY_D bit – which looks at the MISO line going out - is dependent has an odd number of bits. However PARITY_S also looks at the SIF_MISO line coming out and is dependent on a section having an even number of bits.

The PARITY_AC bit in the preferred embodiment looks at the SIF_MOSI line going in.

With reference to the table of section 9.4.5, in the preferred embodiment, the PARITY_AC bit is bit 38 and protects all of bits 0-39. That is, it protects all of the control and address bits.

In the preferred embodiment, the PARITY_S bit is bit 51 and protects all of bits 40 to 51.

If all 0s or all 1s then would get a parity error.

**21. Two separate reset inputs**

The XAP4 has two reset inputs resetb_xap and resetb_sif. It also has a force_stop input. Pulling the resetb_sif low resets the SIF and the debug section of the XAP. This sets RUN_STATE to Running and leaves MODE unchanged. Pulling resetb_xap low resets the rest of the XAP, setting MODE to Supervisor and leaving RUN_STATE unchanged. The processor is stopped whenever the force_stop input is high.

In a further aspect there is provided apparatus comprising a processor and means for resetting the processor, the resetting means being adapted to reset the processor to one of a plurality of modes. Preferably the plurality of modes comprise a normal mode and a debug mode.

Preferably in the debug mode the normal operation of the processor may be running or may be stopped and/or the processor may be operated a single step at a time in response to debug instructions.

Preferably the resetting means is adapted to reset the processor in response to a reset instruction. Preferably the resetting means is adapted to reset the processor to a normal mode in response to pulling resetb_sif low.

Preferably the resetting means is included in, or on the same circuit board or chip as, the processor. Preferably the apparatus includes a dedicated line for communication of the or each reset instruction to the processor. Preferably the apparatus comprises a dedicated line for communication of a normal reset instruction to the processor and a further dedicated line for communication of a debug reset instruction to the processor.

Preferably the apparatus comprises at least one further processor and the resetting means is adapted to reset the at least one further processor. Preferably the resetting means is adapted to reset each of the processor and the at least one further processor by sending a respective reset instruction to that processor.

Preferably the or each reset instruction is specific to a particular processor.

Preferably the interface comprises an interface adapted for communication with the processor and preferably with the at least one further processor. Preferably the interface is adapted to send data to the processor or to the at least one further processor in dependence upon address data, preferably included in the data. The data may include an instruction, for instance the or a reset instruction. Preferably the interface is adapted to receive the data from a control means.

Preferably the interface comprises a SIF, preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. Preferably the control means comprises a SIF master and/or a control computer. Preferably the apparatus comprises means for debugging, or for acquiring data from, the interface and/or the processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 9, and in particular section 9.7, of the XAP4 Specification document.

**22. Processor designed for code density**
**– preferably registers not treated symmetrically by the instructions**

In a further aspect there is provided a processor instruction set which is not regular, but which is tailored to compiler behaviour and/or calling convention.

Further information is provided, for instance, at section 2.3.1 of the Summary of Inventions document.

Preferably registers are not all treated symmetrically by instructions. The instructions may play to the coding conventions used by the compiler. In particular, instruction set support for the compiler's conventions for function entry and exit can achieve significant improvement in code density.

In the preferred embodiment, certain registers are used for certain functions by the C compiler. For instance there may be certain things that may require to be done by register R2 which are not required to be done with register R3. Accordingly, there may be particular instructions for, say, R2 and not for R3.

In the preferred embodiment the first three function arguments are passed in registers R0 to R2 and subsequent function arguments are passed on the stack (ie in memory). Therefore, in the preferred embodiment there are also provided instructions which have preferential support for registers R0 to R2.

Examples of instructions which have preferential support for R0 to R2 in the preferred embodiment include:- pop (see for instance section 8.9.4.2 of the XAP4 Specification document); pop.ret (see for instance section 8.9.4.3 of the XAP4 Specification document); and push (see for instance section 8.9.4.1 of the XAP4 Specification document).

In the preferred embodiment the function return value is passed in register R0 Therefore, in the preferred embodiment there are also provided instructions which have preferential support for register R0.

Examples of such instructions which have preferential support for R0 in the preferred embodiment include:- pop.ret (see for instance section 8.9.4.3 of the XAP4 Specification document).

In preferred embodiments, R6 is used as a link register. This contains the function return address.

In preferred embodiments, R7 is used as a stack pointer. This contains the base address of the stack.

All addresses, in preferred embodiments, are stored in registers as 16 bit byte addresses.

Some instructions may, in practice be used often and other instructions may be used less often. In the preferred embodiments, 16 bit codings are given to those instructions which are used often, and only 32 bit codings are given to those instructions which are used less often.

Preferably there are provided compound instructions, each compound instruction being adapted to carry out a plurality of actions. Thus fewer instruction fetches are required. Examples of such compound instructions in the preferred embodiment include:- push; pop; and pop.ret.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board or chip.

**23. Instructions may be provided in 32 bit or 16 bit forms.**

Preferably each instruction can be 16 bit or 32-bit, freely mixed on an instruction-by-instruction basis. 16 bit instructions require a single memory fetch. 32 bit instructions require a double memory fetch.

Some instructions are only available in 16 bit form. Some instructions are only available in 32 bit form. Some instructions are available in 16 and 32 bit forms.

Instruction decode is performed as soon as the full instruction is available. This will be after the first fetch for 16 bit instructions and after the second fetch for 32 bit instructions.

Preferably a 3 bit field identifies whether the instruction is 16 or 32 bit. Preferably, 7 of the 3 bit values represent 16 bit instructions and the remaining $8^{th}$ value represents 32 bit instructions. This ratio may result in close to optimum code density for a 16 bit processor. This leads to 56k 16 bit instructions and 512k 32 bit instructions.

In a further aspect there is provided an instruction set having equivalent instructions in short and long forms. Preferably instructions which are used frequently and which can be coded in a short form are provided in a short form.

Being able to mix instructions freely on a long and short form basis means that the processor does not need to have distinct instruction length modes, which thus provides a simple programming model for the programmer.

In a further aspect there is provided a processor adapted to process portions of data of variable length, the processor comprising means for determining the length of a portion of data and being adapted to process the portion of data in dependence upon that determination.

Preferably the determining means is adapted to determine the length of a portion of data in dependence upon a length identifier, the length identifier preferably being included in or associated with the portion of data.

Preferably the processor is adapted to process a series of portions of data in turn, the determining means being adapted to determine the respective length of each portion of data in turn, and the processor being adapted to process each portion of data in dependence on the respective determined length.

Preferably the or each portion of data comprises an instruction.

Preferably the or each portion of data may comprise at least one operator and may also comprise at least one argument.

Preferably there is provided an instruction set, preferably for a processor as described herein, comprising an instruction provided in a first portion of data having a first length and an instruction provided in a second portion of data having a second length, the first length being greater than the second length. Preferably the first portion of data comprises an argument and the second portion of data comprises a further argument, the length of the argument of the first portion of data being longer than the argument of the second portion of data. Preferably the first portion of data comprises an operator and the second portion of data comprises the same operator. Preferably each of the first portion of data and the second portion of data comprises a respective length identifier identifying the length of that portion of data. Preferably the first portion of data has a length of 32 bits and the second portion has a length of 16 bits.

In a further aspect there is provided an instruction set comprising a first sub-set of instructions which are provided in both a first form and a second form, and a second sub-set of instructions which are provided in a first form. Preferably each instruction in its first form comprises a portion of data of a first length and each instruction in its second form comprises a portion of data of a second length, the first length being greater than the second length.

Preferably each instruction comprises an operator and/or an argument. Preferably an or each instruction is provided in the second form or not in dependence on the length of a respective operator and/or in dependence on the length of a respective argument and/or in dependence on the likelihood that the instruction will occur in use of the instruction set. Preferably the instruction set is arranged so that commonly used instructions may be coded with arguments and/or operators that are sufficiently short that such commonly used instructions may be provided in both or either the first form and the second form. Preferably each instruction in its first form is a 32 bit instruction and each instruction in its second form is a 16 bit instruction.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.2, 8.3, 8.8 and 8.9, of the XAP4 Specification document. Further information is also provided, for instance, in section 2.3.2 and 2.3.7 of the Summary of Inventions document.

As can be seen from, for instance, sections 8.2 and 8.3 of the XAP4 Specification document, in the preferred embodiment there are provided both 16 bit instructions and 32 bit instructions. At least some instructions are provided in both 16 bit form and 32 bit form, depending on the length of the argument included with the instruction, such instructions generally providing identical functionality in both their 16 and 32 bit forms.

Commonly used instructions in the preferred embodiment are provided both in 16 bit and 32 bit forms (depending on the length of the argument included in the instruction).

In the preferred embodiment, processor memory includes both 32 bit instruction space and 16 bit instruction space. Typically far more 32 bit instruction space is provided than 16 bit instruction space. For instance there may by, say, 8,000 times more 32 bit instruction space than 16 bit instruction space.

In the preferred embodiment, the processor is defined by a 32 bit instruction set and a subset of that which is used frequently is coded in 16 bits. In the preferred embodiment there are around 56,000 different 16 bit instructions and around 500,000,000 different 32 bit instructions.

In the preferred embodiment the 32 bit instruction space overlaps the 16 bit instruction space. Instructions may be identical in their 16 bit form and 32 bit forms but they may they have different (i.e. smaller in the case of the 16 bit form) argument ranges.

Preferably the instruction set is arranged so that roughly 80 to 90% of operations carried out are from 16 bit instructions (with regard to this statement at least, an operation could be considered to be, say, adding 13 to register R5, whilst, say, adding 14 to register R5 would be considered to be a different operation).

In the preferred embodiment the compiler produces assembler code which does not indicate whether an instruction is 16 bit or 32 bit. For each piece of assembler code, the assembler then determines whether there is underlying 16 bit code, or only underlying 32 bit code, available.

As discussed, instructions have a minimum size of either 16 bits or 32 bits in the preferred embodiment. Preferably some 16 bit instructions also have a 32 bit form.

Section 8.8 of the XAP4 Specification document shows code numbers allocated to various instructions in the preferred embodiment.

Code numbers may be allocated in a way that makes the decode easier.

In the preferred embodiment, there are 3 bits in the primary op field and there are 8 primary op codes. In the preferred embodiment each primary op code can take a 13 bit argument. The 13 bit argument may be split into different fields.

In the embodiment shown in section 8.8, op codes 0 to 6 are 16 bit instructions while op code 7 is for 32 bit instructions.
Details of extended op code instructions in the preferred embodiment are provided in sections 8.8. of the XAP4 Specification document.

Preferably, some instructions are provided which have a separate prime opcode for each register (eg mov.p). This allows coding long immediates in single instructions.

## 24. Little endian/ big endian

In known systems, byte addresses are generally used to address memory for a range of processor sizes (for instance, 8 bit, 16 bit and 32 bit processors).

In known systems, processors include branch instructions to branch to the location of the next instruction to be executed. The address used in the branch instruction is normally the low address of the instruction being branched to (which is significant if the instruction contains more than one byte). That low address is commonly referred to as the low byte. In little endian systems the low byte represents the bottom 8 bits of the instruction (or data word). In big endian systems the low byte represents the top 8 bits of the instruction (or data word).

In a further aspect there is provided a processor having a variety of instruction sizes determined on an instruction by instruction basis.

That enables higher code density than would be achieved with a single fixed instruction size whilst still being simple for the programmer to use.

Preferably each instruction comprises at least one instruction size bit to indicate the instruction size.

Preferably the at least one instruction size bit is provided in the lowest byte, preferably in the lowest bit position, in a little endian architecture processor.

Alternatively, the at least one instruction size bit is provided in the highest byte, preferably in the highest bit position, in a big endian architecture processor.

Thus it is ensured that the at least one location size bit is always in the byte address branched to. Preferably this is the lowest byte address of the instruction.

Preferably unaligned memory access is supported. A portion of 32 bit data may have any address.

In a further aspect 32 bit instructions and/or 16 bit instructions are provided, and there is also provided a bit included in an instruction which indicates whether the instruction is a 16 bit instruction or a 32 bit instruction.
In a further aspect 32 bit instructions and/or 16 bit instructions are provided, and 3 bits are provided in an instruction to indicate whether the instruction is a 16 bit instruction or a 32 bit instruction. This enables finer resolution for the ratio of 16 bit to 32 bit instruction space.

In a further aspect there is provided a processor having an instruction set in which the ratio of 16 to 32 bit instructions is greater than 2:1 and less than 32:1 and preferably 7:1.

In a further aspect there is provided a method of processing data comprising a plurality of bits arranged in a series, the plurality of bits comprising at least one bit representing data of a first type and at least one bit representing data of a second type, the at least one bit representing data of a first type being provided in a selected position or range of positions in the series.

Preferably the first type data comprises argument data, preferably at least one immediate. Preferably the second type data comprises operating code. Preferably the data comprises 32 bit or 16 bit data. Preferably the first type data is provided from a particular bit upwards in series of bits. Preferably the first type data is provided from the 8$^{th}$ bit (bit 7 when counting from zero) upwards in a series of bits. Alternatively the first type data may be provided from a particular bit downwards in a series of bits (for instance if using a big endian architecture).

In an embodiment the data is in the form of an instruction.

Preferably the first type data is processed before, and/or independently of, the second type data.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. The processor may form part of an ASIC or FPGA and preferably the interface also forms part of the ASIC or FPGA. Preferably the processor and the interface are on the same circuit board.

In a further aspect there is provided a XAP4 processor adapted to run off the shelf software, for instance GNU software, such as the GNU C or C++ compiler, and/or adapted to communicate using TCP/IP protocol, and/or ethernet protocol and/or Bluetooth.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.2 and 8.3, of the XAP4 Specification document. Further information is also provided, for instance, in section 2.1.2 of the Summary of Inventions document.

## 25. Instruction set characteristics

In a further aspect there is provided an instruction set for a processor, the instruction set having a common format which applies to at least some and preferably all of the instructions in the instruction set.

Thus a simple instruction decode is provided, which leads to low power consumption and a smaller hardware implementation.

Preferably the common format provides a respective fixed position where each of the following register selectors, if present, appears in any instruction to which the common format applies:- a destination register selector (Rd in the preferred embodiment); a primary source register selector (Rs in the preferred embodiment); a secondary source register selector (Rt in the preferred embodiment); a base address register (Ra in the preferred embodiment); and an index address register (Rx).

Preferably the common format provides a fixed position where the least significant bit of an immediate, if present, appears in any instruction to which the common format applies.

In a further aspect there is provided an instruction set for a processor comprising a plurality of instructions, each instruction comprising a respective portion of data and each of at least some of the instructions comprising a respective sub-portion of data comprising a plurality of bits and at least one sign bit.

Preferably at least one of the sub-portions of data is representative of a register selector. In the preferred embodiment the register selector has 3 bits to select one of 8 registers.

Preferably at least one of the sub-portions of data is representative of an argument, preferably an immediate.

Preferably at least one, and preferably all, of the sub-portions of data comprise 16 bits. Preferably at least one, and preferably all, of the sub-portions of data, comprise 15 bits and one sign bit.

Preferably at least one of, and preferably all of, the instructions further comprises an address, preferably a register address, preferably in a sub-portion of data. Preferably at least one of, and preferably all of, the instructions further comprises an operator, preferably in a sub-portion of data.

Preferably at least one of, and preferably all of, the instructions comprise a plurality of sub-portions of data, at least some of the respective sub-portions of data for each instruction being of a fixed length or a multiple of a fixed length. Preferably the fixed length is 3 bits. Preferably the sub-portions of data for at least some of the instructions are aligned with corresponding sub-portions of data for others of the instructions.

In a further aspect there is provided an instruction set for a processor comprising a plurality of instructions, each instruction comprising a respective plurality of sub-portions of data. Preferably for each instruction at least some, and preferably all of the respective portions of data are of a fixed length or a multiple of a fixed length. Preferably the fixed length is 3 bits. Preferably the respective sub-portions of data for at least some of the instructions are aligned with corresponding sub-portions of data for others of the instructions.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. The processor may form part of an ASIC or FPGA.

Preferably, 16-bit immediates are provided, which allow both 16-bit signed (for instance -32768...+32767) and unsigned (for instance 0...65535) value ranges to be encoded in a single immediate. This enables an implementation with smaller hardware and lower power consumption.

Preferably processor instructions sign extend or zero extend immediates to represent full word length. In the preferred embodiment, full word length is 16 bits.

A 6 bit argument may be used to represent 2 register selectors. A 9 bit argument may be used to select 3 register selectors.

Features of the preferred embodiment are described with reference to, for instance, section 8, and in particular sections 8.2 and 8.3, of the XAP4 Specification document. Further information is also provided, for instance, in section 2.3.6 of the Summary of Inventions document.

It can be seen from the tables of sections 8.2 and 8.3 of the XAP4 Specification document that both the 16 bit and 32 bit instructions in the preferred embodiment are aligned in 3 bit sections.

In the preferred embodiment, arithmetic and logic instructions are provided in two formats (as described in section 7 of the XAP4 Specification document). In Register, Register, Immediate (RRI) format there is one register input, one immediate input, and one register output. In Register, Register, Register (RRR) format there are two register inputs and one register output.

Typically in the preferred embodiment, particular register addresses occur at the same place in different instructions. For instance it can be seen from section 8.2 and 8.3 that the destination register (Rd) address generally is found at bits 15:13. Similarly, each of the first source register (Rs), second source register (Rt), base address register (Ra) and index register (Rx) addresses are generally found at a respective particular location in the instructions.

In the preferred embodiment, memory is only accessed through a load, from memory to register, or a store, from register to memory, or multiples of such operations, or through push and pop instructions, which are other multi-cycle instructions associated with the stack or with the swap instruction.

Preferably, arithmetic or logic instructions such as AND, OR, ADD, SUBTRACT, MULITPLY do not act on memory. They either take two registers as an input or a register and an immediate and then the output is always a register.

## 26. Move Multiple instructions (movm, movm.x)

These features are implemented in certain embodiments of the invention (such as XAP3), and not implemented in other embodiments (XAP4 or XAP5).

## 27. Instruction mnemonics

In a further aspect there is provided a set of instructions for a processor, each instruction having a respective mnemonic, the mnemonics being organised hierarchically.

Such hierarchical organisation makes it easier for a programmer to learn and use the set of instructions.

Preferably the set of instructions comprises a set of assembler instructions. Preferably each of the set of instructions comprises a respective series of components separated by delimiters. Preferably each of the series of components is ordered by the significance of each component. Preferably each delimiter is a dot. Preferably the mnemonics are case sensitive. Preferably the mnemonics are all lower case.

Preferably the instruction mnemonics are of the form base.parameters.type.size.

Preferably the format of assembler identifiers is such as not to restrict the names available in C source code. For example, in the preferred embodiment, %r1 and %r7 are assembler identifiers, and they could not in any event be used in C.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. The processor may form part of an ASIC or FPGA.

Features of the preferred embodiment are described with reference to, for instance, section 2, and in particular section 2.5, and section 8, in particular section 8.4, of the XAP4 Specification document. Further information is also provided, for instance, in section 2.4.2 of the Summary of Inventions document.

In the preferred embodiment the instruction mnemonics are of the form base.parameters.type.size. All mnemonics include base. Parameters, type and size are optional but where they exist they are in the order base.parameters.type.size. Examples of such instruction mnemonics include:-

add.i
add.r
add.c.r
cmp.c.r
cmp.8c.r
cmp.16xc.r
or.i
or.h
or.r
bra.a
bra.a.4
bra.a.2
bra.p
mov.i
mov.r
mov.32.r
mov.p
ld.i
ld.8z.p
ld.r
st.8.i
stz.r

In the preferred embodiment there is no instruction name space pollution. Names used in C do not need to be converted as they go through the assembler.

Preferably reserved word features used in assembler would not be valid in C. Preferably any reference to a register or registers is prefixed by a percentage. Preferably immediates are prefixed with a hash. Preferably the assembler instructions are case sensitive. Preferably reserved words in assembler are lower case.

## 28. High immediates and down-extending

These features are implemented in certain embodiments of the invention (such as XAP3), and not implemented in other embodiments (XAP4 or XAP5).

**29. push instruction**

In a further aspect there is provided an instruction (for instance push in the preferred embodiment) which writes multiple registers to a stack, allocates space for local (automatic) variables, and updates a stack pointer register accordingly.

In the preferred embodiment, the push instruction pushes all selected registers from R3 to R6 onto the stack (see for instance 8.9.4.1 of the XAP4 Specification document). This saves many instructions at the beginning of a function for saving call-save registers on the stack. This reduces power consumption.

In the preferred embodiment, the push instruction assumes that R7 is the stack pointer.

Efficient function entry and exit is essential for good code density. In the preferred embodiment, efficient function entry is provided by the push instruction.

Preferably the processor comprises an 8 bit, 16 bit, 32 bit, or 64 bit processor. Preferably the processor comprises a XAP1, XAP2, XAP3, XAP4 or XAP5 processor. The processor may form part of an ASIC or FPGA.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.9.4.1, of the XAP4 Specification document. Further information is also provided, for instance, in section 2.3.3.3 of the Summary of Inventions document.

## 30. pop, pop.ret instructions

In a further aspect there is provided an instruction for a processor (for instance a pop or pop.ret instruction in the preferred embodiment) which loads at least one register from memory, via a stack pointer, removes zero or more words of automatic storage (for instance as allocated by an instruction according to section 29 above, such as a push instruction in the preferred embodiment), and updates the stack pointer.

Preferably the instruction takes an additional argument (preferably a register or small immediate) and stores that argument in a register (for instance in R0 in the preferred embodiment), and updates condition flags as if that value was compared with 0. Preferably the instruction (for instance the pop.ret instruction in the preferred embodiment) then executes a return via another register (for instance the Link Register R6 in the preferred embodiment). This reduces the number of instructions, thus reducing memory size and power consumption.

Efficient function entry and exit is essential for good code density. In the preferred embodiment, efficient function exit is provided by the pop and pop.ret instructions.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.9.4.2 and 8.9.4.3, of the XAP4 Specification document. Further information is also provided, for instance, in section 2.3.3.4 of the Summary of Inventions document.

## 31. Load Multiple (ldm) / Store Multiple (stm) instructions

These features are implemented in certain embodiments of the invention (such as XAP3), and not implemented in other embodiments (XAP4 or XAP5).

## 32. Movm arguments encoded as 5 bit fields

These features are implemented in certain embodiments of the invention (such as XAP3), and not implemented in other embodiments (XAP4 or XAP5).

**33. Register R0 = constant 0**

These features are implemented in certain embodiments of the invention (such as XAP3), and not implemented in other embodiments (XAP4 or XAP5).

## 34. Uneven register visibility in instruction set

In a further aspect uneven register visibility in instructions, for instance in 16 bit instructions is provided to optimise for code density.

In a further aspect there is provided a processor and associated instructions, in which registers of the processor are not all treated symmetrically by the processor instructions. Preferably the use of registers is tailored to the coding conventions used by the compiler. Certain registers are used for certain functions by the compiler.

In the preferred embodiment, the design of the 16-bit instruction set has been further optimised to be even closer to the output of the compiler; for example, some 16-bit instructions allow access to R0-R6 but not to R7 due to register usage behaviour of the compiler.

The preferred embodiment has been designed for code-density.

There are certain things that it may be desired to do with, say, register R3 which it is not desired to do with, say, register R2. Accordingly, in the preferred embodiment there are certain instructions for R3 and not for R2.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.2 and 8.8 of the XAP4 Specification document. Further information is also provided, for instance, in section 2.3.8 of the Summary of Inventions document.

## 35. sleepsif and sleepnop instructions enable external debugging

In a further aspect there is provided an instruction for a processor adapted to communicate via an interface, the instruction when processed by the processor causing communication from the interface to the processor and/or from the processor to the interface to be stopped or not allowed, preferably for a fixed period and/or until a further instruction is processed.

In a further aspect there is provided an interface for communication with a processor, the interface being responsive to an instruction from the processor to stop communication with the processor and/or to not allow communication with the processor and/or to not allow transmission of data from the interface to the processor, preferably for a fixed period and/or until receipt of a further instruction.

A sleepsif instruction may allow external debugger access to processor registers and memory while the processor is in the low-power sleep state. A sleepnop instruction may prevent SIF accesses. These two instructions allow an application programmer to control precisely when a SIF has access to the system's resources.

Distinct sleepnop and sleepsif instructions may be provided. Control from the slave end is thus provided, enabling it to ensure that the SIF is quiet, for instance allowing the slave to take readings.

Preferably the processor has two states: a "sleeping" state and an "awake" state. In the sleeping state at least part of the operation of the processor may be suspended, for example by suspending the execution of any further instructions, which may reduce the power consumed by the processor. In the awake state, the processor may be capable of a greater degree of operation than is possible in the sleeping state: for example the processor may execute instructions that are not available in the sleeping state. Thus, the greater degree of activity permitted in the awake state results in the processor

consuming more power whilst in the awake state than whilst in the sleeping state.

The sleeping state may comprise two sub-states: a sleepsif state and a sleepnop state. In the sleepsif state the processor may be capable of performing functions related to debugging, and in particular a SIF Module may be fully operable. However, in the sleepsif state, other functionality of the processor may not operable and in particular the processor may not operable to execute instructions unrelated to debugging such as those of User Mode programs.

In the sleepnop state, the processor may not be operable to execute any instructions: more particularly, neither instructions related to debugging nor instructions unrelated to debugging may be executed in the sleepnop state. Thus, in the sleepnop state, the SIF Module may be inoperable. The provision of the sleepnop state can further reduce the power consumption of the processor relative to that of the sleepsif state and the processor awake state. Additionally, by deactivating the SIF Module in the sleepnop state, electrical noise that may be produced by the SIF Module may be reduced, which may allow more accurate measurements to be made by an analogue to digital converter that is incorporated in an electrical circuit comprising the microprocessor.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular sections 8.4 and 8.9.12, and section 9, in particular section 9.3, of the XAP4 Specification document, and in the SIF Definition document. Further information is also provided, for instance, in section 2.3.9 of the Summary of Inventions document.

## 36. Support of position independent code

Support of position-independent code (PIC) requires computing addresses from some defined reference point. In a further aspect there is provided a reference point: a Program Counter (PC) for code, constant and global data references.

In the preferred embodiment the compiler generates PC relative code, constants and global variables. Thus providing position independent executables and data.

In a further aspect there is provided an instruction set for computing absolute addresses at run time from the PC relative instructions.

In the preferred embodiment the PC relative instructions include mov.p (see for instance section 8.9.5.1 of the XAP4 Specification document). Other PC relative instructions include ld.p, st.p, and stz.p (see for instance section 8.9.3 of the XAP4 Specification), bra.p and bsr.p (see for instance section 8.9.1 of the XAP4 Specification document) and all conditional branch instructions (see for instance section 8.9.2 of the XAP4 Specification document).

These instructions minimise run time fixups for position independent code thereby reducing load time and power consumption.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular section 8.9.5.1 of the XAP4 Specification document. Further information is also provided, for instance, in section 2.3.10 of the Summary of Inventions document.

## 37. Conditional Set instructions

These features are implemented in certain embodiments of the invention (such as XAP3), and not implemented in other embodiments (XAP4 or XAP5).

## 38. Breakpoint debug support

Processor code may have debug registers with a count conditional to trigger a breakpoint after some numbers of executions of the target instructions. This simplifies debugging of say, loops.

As this is part of the core, and not an external module, it facilitates common debug environment both for development and in-field debugging. Especially for debugging user mode code by an operating system.

There may be provided hardware breakpoint registers in the processor core which support software-implemented multi-stage breakpoint. The breakpoint registers may use in-core debug registers, and/or a debug module within OS, and/or a textual or graphical interface to specify the breakpoint state.

There is provided a SIF-Ethernet pod that allows for remote debugging of applications over the internet.

In a further aspect there is provided a processor core having a debugging means.

Preferably the debugging means is integral with the core. Preferably the debugging means is fully contained within the core.

A further aspect provides a debugging means. A yet further aspect provides a debugging means for a processor core.

Preferably the core is adapted to execute a computer program. Preferably the debugging means is adapted to facilitate the debugging of a computer program. For example, the debugging means may trigger a predetermined instruction (or sequence of instructions) upon the occurrence of one or more specified events in the program to be debugged, thereby enabling the operation of the computer program to be monitored by a programmer or engineer.

Preferably the debugging means is adapted to enable the execution of a debug processing step. For example, execution of the debug processing step may include the execution of one or more instructions, subroutines and/or computer programs.

Preferably the debugging means includes a breakpoint address register. Preferably the breakpoint register is adapted to reference (or to store information relating to) the address of a memory location. Preferably the breakpoint register is adapted to reference a memory location external to the processor core.

Preferably the debugging means is adapted to enable the execution of the debug processing step when predetermined processing is performed in relation to the address referenced by the breakpoint address register.

Preferably the predetermined processing includes one or more of:

i) The address referenced by the breakpoint address register being accessed a predetermined number of times

ii) A predetermined data value (or a range of predetermined data values) being read from and/or written to the address referenced by the breakpoint address register

iii) Enable bits for breaks to occur when the specified address is the instruction and/or a data read and/or a data write

Preferably the debugging means is adapted to enable the execution of the debug processing step when the address referenced by the breakpoint address register has been accessed a predetermined number of times.

Preferably the debugging means is adapted to enable the execution of the debug processing step when the address referenced by the breakpoint address register has been accessed a predetermined number of times by a selected access mode. Preferably the access mode is selectable from the following: write (relating to the writing of data to memory); read (relating to the

reading of data from memory); or fetch (relating to the reading of an instruction from memory).

Preferably the debugging means comprises a breakpoint control register for storing information relating to the selected access mode.

Preferably the debugging means comprises a count register for storing information relating to the number of times that the address referenced by the breakpoint address register has been accessed.

Preferably the debugging means is adapted to enable the execution of the debug processing step when a predetermined data value (or a range of predetermined data values) is read from and/or written to the address referenced by the breakpoint address register.

Preferably the debug processing step is selectable from a plurality of such debug processing steps.

Preferably the debug processing steps include one or more of the following:

i) generating an exception

ii) halting the processor.

Preferably the processor comprises a break enable flag for storing information relating to the selected debug processing step.

Preferably the core comprises a plurality of operating modes. Preferably the operating modes are selectable. Preferably each of the operating modes has a respective privilege level. For example, the core may have one or more of the following operating modes: a privileged Supervisor Mode; a privileged Interrupt Mode; a privileged NMI (non-maskable interrupt) mode; and an unprivileged User Mode.

Preferably the debugging means comprises a register that is not accessible by a program operating in an unprivileged operating mode. Preferably the

debugging means comprises a register to which data cannot be written by a program operating in an unprivileged operating mode. Preferably the debugging means comprises a register from which data cannot be read by a program operating in an unprivileged operating mode.

A software debugger may be used to debug a program being executed on a processor that does not comprise a hardware debugger, or that comprises a hardware debugger without this functionality. In the case of a software debugger, the registers and/or flags described herein may be implemented in software as variables.

Features of the preferred embodiment are described with reference to, for instance, section 5, and in particular section 5.5, and section 9, of the XAP4 Specification document.

Certain features of the preferred embodiment, and of possible alternative embodiments are now described in more detail, purely by way of example.

In the preferred embodiment, the processor executes a sequence of instructions comprising one or more programs. Such programs require testing, inspection and verification during development, and testing and correction when the product is in use.

The use of multiple processing units in a single system, or even a single integrated circuit, requires the support of an integrated and uniform debug and verification system to make it feasible to develop reliable and robust software within acceptable timescales.

In many applications of the processor the program is held in memory which is difficult or impossible to change once the product has been assembled.

A means is described in which the processor core itself can stop and examine the state of one or more programs currently executing on the processor. The invention is part of the core of the processor, and as such is present in all

variations of the processor core. This facilitates a debug infrastructure (debug software, external debug interface, remote debug software, communication protocols, etc) common to all instances of the processor. This reduces development costs through design re-use, and easily supports the debugging of multi-processor designs through a common debug interface shared among the processor cores.

A processor includes a set of registers (the processor interface) and supporting hardware gates (the implementation).

Certain registers, known as breakpoint registers, can contain information relating to breakpoints. A breakpoint is a point of execution of a program that is characterised by either the address of an instruction in the program (or one program of several), or the address of a location in memory at which the program may access. Upon encountering a breakpoint execution may stop and control of the processor be taken by other software (e.g. operating system, debug executive, external debugger, etc).

These breakpoints may additionally be qualified by either or both of two predicates (conditions):

a) a count of the number of times the instruction may execute, or memory be accessed, before causing a break;
b) a predetermined data value (or one or more ranges of predetermined data values) transferred to or from memory external to the processor core.

In the preferred embodiment, the processor comprises a debug system supporting eight breakpoints, four of which are further qualified with a count predicate, and in which two of these count-qualified breakpoints are yet further qualified by data predicates.

In the preferred embodiment eight breakpoint registers and a breakpoint control register to describe the behaviour of four breakpoints are provided. The eight breakpoint registers are named BRK0 through BRK3, BRK2COUNT through BRK3COUNT, BRK3DATA, and BRK3MASK. The breakpoint control

register is named BRKE. A single bit in the processor's FLAGS register specifies if breakpoints should occur or be ignored.

The breakpoint address registers, BRK0 through BRK3, respectively describe a 16-bit wide address for each of the four breakpoints. These addresses can either be the address of an instruction or the address of a unique memory element external to the processor core.

The count registers, BRK2COUNT through BRK3COUNT, respectively describe a 16-bit wide count predicate corresponding to breakpoints 2 through 3 (as specified in the registers BRK2 through BRK3). A break occurs if an address match happens (that is, the address specified in a breakpoint address register matches the address to/from which an instruction or data is to be read, written or stored) and if the value of the count register corresponding to that breakpoint address register is zero. If an address match occurs, but the value of the corresponding count register is not zero, the value of the count register is decremented by one and no break occurs. The count register preferably cannot be decremented below zero: once it reaches zero, it remains at zero until a new value is written to it.

The default value of the count registers is zero, such that normal behaviour is the same as for the first two breakpoints. That is, when the value in a count register is zero, the corresponding breakpoint behaves like a simple breakpoint.

The BRK2DATA and BRK2MASK registers further specify a data qualifier on the corresponding breakpoint, as specified in register BRK2. These registers operate such that a break occurs when a data value read from (or written to) memory at the location specified in register BRK2 satisfies the data qualifier. It will be appreciated that the "data value" can preferably be either an instruction or data. The data qualifier is specified for each of the 16 bits in the data value. For each bit, the qualifier is satisfied if either

a) the corresponding bit in the mask register, BRK2MASK, is a '1' and the corresponding bit in the data register, BRK2DATA, matches (that is, is equal to) the corresponding bit in the data value, or
b) the corresponding bit in the mask register is '0'.

Thus, by setting a bit in the mask register to '1', the occurrence of a break is conditional upon the bit of the data value that corresponds to that bit of the mask register being equal to the corresponding bit of the data register. Alternatively, by setting a bit in the mask register to '0', a "don't-care" condition is selected, and the occurrence of a break is not conditional upon the value of the corresponding bit of the data value.

The BRK3DATA and BRK3MASK registers operate similarly to the BRK2DATA and BRK2MASK registers, but in respect of the breakpoint specified in register BRK3.

Thus, a break can be caused when a predetermined data value is read from and/or written to a predetermined location in memory. By setting a plurality of bits of the mask register to '0', a break can be caused when a data value belonging to a set of predetermined data values is read from and/or written to memory. For particular values of the mask register, the set of predetermined data values can include one or more ranges of consecutive values.

The default value of the mask and data registers is zero, such that normal behaviour is the same as for the first two breakpoints. That is, when a mask register is set to zero, the corresponding breakpoint behaves like a simple breakpoint, since all bits of the mask register treated as "don't-cares" and the breakpoint will occur irrespective of the data value being read from and/or written to memory.

The breakpoint control register BRKE is a 16-bit wide register. It is split into four 4-bit wide fields, where each field corresponds to one of the four breakpoints.

Each field has three control bits and one unused bit; the fourth bit is left for future debug facilities, for example. The three control bits specify whether a breakpoint should occur:

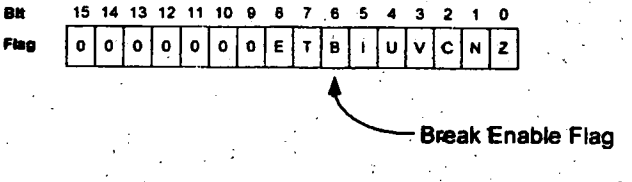
a) if an instruction writes to the address matching the corresponding breakpoint address;
b) if an instruction reads from the address matching the corresponding breakpoint address;
c) if an instruction has been fetched for execution from the address matching the corresponding breakpoint address.

Finally, there is a breakpoint enable bit in the FLAGS register. If this bit is set and the processor is in the lowest privileged execution mode then breakpoints will cause an exception to occur to allow the debug software to take control of the processor and monitor and/or modify the state of the program or programs. Thus, by setting the breakpoint enable bit, an exception is generated rather than halting the processor.

Features of a debug system for a processor will now be described in detail.

**Flags register**

The FLAGS register describes or indicates various signals within the processor core. The 'B' flag controls whether breakpoints cause breaks in execution or are ignored. The diagram below indicates the position of the 'B' flag within the FLAGS register.



**BRKE Register**

The BRKE register is used to control the behaviour of the four hardware breakpoints described above.

The bits in the BRKE register are described in the table below.

| *Bit(s)* | *Flag(s)* | *Name* | *Description* |
|---|---|---|---|
| 0, 4, 8, 12, | W0, W1, W2, W3, | Write | If set, an instruction making a memory write to the address matching the address (all 16 bits) will halt the processor after the instruction has executed. The Program Counter (PC) will point at the next instruction to be executed. |
| 1, 5, 9, 13, | R0, R1, R2, R3, | Read | Similar to above, but for memory reads. |
| 2, 6, 10, 14, | E0, E1, E2, E3, | Execute | If set, an instruction fetch from the address matching the address (all 16 bits) will halt the processor before the instruction is executed. The PC remains at the address of the instruction that caused the break. |
| 3, 7, 11, 15, | Not used | | |

The BRKE register is one of a number of special registers that are only accessible to programs executing in one of the privileged execution modes. This prevents non-privileged programs from monitoring or modifying these registers and thereby monitoring or modifying their own debug or test conditions.

Providing privileged modes access to the breakpoint registers allows Supervisor Mode debuggers to be implemented for debugging User Mode tasks without needing to stop the processor. This can allow, for example, the main operating system to remain running when debugging User Mode applications.

**Breakpoint Control Registers**

The breakpoint registers can be written and read by privileged modes using special instructions. These instructions can only be executed by programs executing in one of the privileged execution modes. This prevents non-privileged programs from monitoring or modifying their own debug or test conditions.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| %brk0 | 000 | BRK0 | Breakpoint 0 |
| %brk1 | 001 | BRK1 | Breakpoint 1 |
| %brk2 | 010 | BRK2 | Breakpoint 2 |
| %brk3 | 011 | BRK3 | Breakpoint 3 |
| %brk2count | 100 | BRK2COUNT | BRK2 counter |
| %brk3count | 101 | BRK3COUNT | BRK3 counter |
| %brk3mask | 110 | BRK3MASK | BRK3 data mask |
| %brk3data | 111 | BRK3DATA | BRK7 data match |

**Breakpoint Control Function**

Each breakpoint corresponds to logic gates which implement the control function for the breakpoints. We describe the logic function for breakpoint 3; the other breakpoints are described in a similar fashion with simplifications for breakpoints BRK0 through BRK2 which do not have data predicates, and BRK0 through BRK1 which also do not have count predicates.

The breakpoint BRK3 is triggered if the following four conditions are satisfied:

a) the value on the address bus is exactly the same as the value in the BRK3 register;
b) the mode (read/write/execute) of the memory access cycle is as specified in the corresponding bits in the BRKE register;
c) the value of the BRK3COUNT register is zero;
d) for each bit of the data bus, the corresponding bit in the BRK7MASK register is either 0, or the bit of the data bus matches the corresponding bit in the BRK3DATA register.

If conditions (a) and (b) are met but condition (c) is not met (i.e. the value of BRK7COUNT is not zero) then the value stored in BRK3COUNT is decremented by one.
The behaviour of breakpoints BRK2 is as above but without condition (d).
The behaviour of breakpoints BRK0 through BRK1 is as above but without conditions (c) and (d).

**Debug Interface**

There are two interfaces to the debug registers: via privileged software running on processor and via an external debug system through a "SIF" hardware debug interface. The operation of a SIF interface in particular embodiments is described in more detail in United Kingdom Patent 2 294 137 B, the contents of which are herein incorporated by reference.

Both interfaces can be operated in parallel. This allows, for instance, debug software operating on the processor core to debug an application, with an external debug system monitoring the status of breakpoint counters in real time.

Internal access to the debug registers is via privileged instructions. Such instructions can only be executed by software executing in Supervisor or Interrupt or NMI modes. In non-privileged modes the processor will not execute these instructions.

## 39. Block Instructions

In a further aspect, there is provided an instruction for a processor which executes a single and/or multiple similar sub-instructions repeatedly.

Preferably, the repeatedly executed sub-instructions are in the form of copy and/or store instructions.

Preferably, the instruction is interruptible. More preferably, on interrupt, a current sub-instruction is completed before processing the interrupt routine. The benefits of making such instructions interruptible is that interrupt latency ins reduced. Some instructions are not interruptible so that behaviour is always atomic and deterministic.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular, section 8.9.9 of the XAP4 Specification.

In particular, the instructions blkcp.r, blkcps.r, blkst.r and blkst.8.r are provided.

## 40. Store Zero Instructions

In another aspect, there is provided a processor which is provided without a constant zero register. In this way, compiled code density may be improved.

Preferably, the processor comprises between 4 and 12 normal registers, and more preferably 8 normal registers. None of these registers is a dedicated constant zero.

Preferably, the processor comprises an instruction adapted to store a zero value to a specified memory location.

This important feature may be provided independently. According to a further aspect, there is provided a processor which comprises an instruction adapted to store a zero value to a specified memory location.

Preferably, the instruction operates in a similar fashion to a normal store instruction. More preferably, the only difference between the instruction and a normal store instruction is that a register specifying a source operand (Rs in the XAP4 preferred embodiment) is not required.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular, section 8.9.3.4 of the XAP4 Specification.

## 41. Instruction Decode Format

In another aspect, there is provided a processor, which comprises an instruction set having a substantially uniform encoding.

Preferably, the instructions are in either 16 bit or 32 bit format, and the encoding style is substantially uniform between 16 bit and 32 bit instructions.

In this way, the same processor hardware may be used to encode different instructions.

Preferably, the encoding style for 16 bit instructions follows a 3,3,3,4,3 bit pattern, with each group of bits representing a register address and or an immediate. Similarly, the encoding style for a 32 bit instruction will follow a 16, 3,3,3,4,3 pattern. Thus, it is apparent that in this preferred embodiment the encoding of 16 and 32 bit instructions is largely the same, the only difference being that a 32 bit instruction has the ability to address more memory and/or to handle larger immediates.

Preferably, the first 3 bits of each instruction are in the form of an op code, which indicates whether the instruction is a 16 bit or 32 bit instruction. In the case of a 32 bit instruction two fetch operations are performed before executing the instruction.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular, section 8.2 and 8,3 of the XAP4 Specification.

## 42. Use R7(stack pointer) holes for other instructions

One of the processor registers is used as a stack pointer. Preferably, this register is R7. It is not meaningful for all instructions to operate upon the stack pointer register. This means that holes are left in the instruction set space for the register value of the stack pointer. These holes may be used for other instructions.

In another aspect, there is provided a processor having an instruction set in which the register normally used as a stack pointer can be used to represent a different instruction.

Thus, when the stack pointer is not required the space left in the instruction set may be used for another function.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular, section 8.8.1, 8.8.2, 8.8.3 and 8.8.4 of the XAP4 Specification.

## 43. PC relative loads and stores (ld.p and st.p)

In another aspect, there is provided at least one instruction for a processor, which is suitable for operating on a memory location specified, at least in part, by a register containing the address of the next instruction to be executed by the processor.

Preferably, the instruction includes both a read and a write form, in which case the instruction either reads data from or writes data to a memory location specified in part by the register. In a preferred embodiment (XAP4), the read instruction is in the form of a load instruction (ld.p) and the write instruction is in the form of a store instruction (st.p). In this way code density may be improved.

Preferably, the register is in the form of a program counter (PC), in which case the data presently stored in the PC is interpreted to be a memory address for the read or write instruction.

Preferably, the instruction comprises means for altering the memory address. More preferably, the instruction comprises means for altering the data presently stored in the program counter prior to performing the instruction.

Preferably, the instruction comprises means for adding an offset to the data stored in the program counter thereby to generate an address to read from or write to.

Preferably, both the offset and PC are interpreted as byte addresses.

These instructions are used for global data which can be variables and constants. Such instructions improve code density.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular, section 8.9.3.3 of the XAP4 Specification.

## 44. Shared hardware for div, rem and divrem instructions

In another aspect, there is provided a processor, having a set of arithmetic logic instructions in which the same hardware is used to implement each of the instructions.

Preferably, the instructions are in the form of division and remainder instructions.

In a preferred embodiment (XAP4) they include the division, division and remainder and remainder instructions (div, rem and divrem).

In a preferred embodiment there are 24 instructions supported by a single hardware block. This means that the hardware is much smaller than it would be if implemented separately for each of the 24 instructions.

Features of the preferred embodiment are described with reference to, for instance, section 8, in particular, section 8.9.6.4 of the XAP4 Specification.

The invention extends to methods and/or apparatus substantially as herein described with reference to the accompanying drawings.

Any feature in one aspect of the invention may be applied to other aspects of the invention, in any appropriate combination. In particular, method aspects may be applied to apparatus aspects, and vice versa.

The invention also provides a computer program and a computer program product for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein, and a computer readable medium having stored thereon a program for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein.

The invention also provides a signal embodying a computer program for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein, a method of transmitting such a signal, and a computer product having an operating system which supports a computer program for carrying out any of the methods described herein and/or for embodying any of the apparatus features described herein.

Furthermore, features implemented in hardware may generally be implemented in software, and vice versa. Any reference to software and hardware features herein should be construed accordingly.

Preferred features of the invention are described in the following documents:- SIF (Serial Interface) Definition document (that document being labelled C4862-TM-005 v3.0, and having 74 pages, the pages being numbered from page 1 of 74 to page 74 of 74); document entitled XAP4 Specification (that document being labelled C7432-S-001 v2.0, and having 131 pages, the pages being numbered from page 1 of 131 to page 131 of 131); document entitled xSIF API (that document being labelled C4862-TM-016 v0.10 and having 43 pages, the pages being numbered from page 1 of 43 to page 43 of 43); document entitled XAP4 Programmer's Manual (that document being labelled

C7432-UM-002 v1.3 and having 224 pages, the pages being numbered from 1 of 2224 to 224 of 224); document entitled Summary of Inventions (that document being labelled C7245-TM-004 v0.7 and having 5 pages, the pages being numbered from page 3 of 10 to page 7 of 10); document entitled XAP4a Hardware Reference Manual (that document being labelled C7432-UM-011 v1.4 and having 47 pages, the pages being numbered from 1 of 47 to 47 of 47); and document entitled XAP4a Instruction Set Quick Reference Card (that document being labelled C7432-UM-005 v1.11 and having 4 pages, the pages being numbered from 1 of 4 to 4 of 4)

It will be understood that the present invention is described below purely by way of example, and modification of detail can be made within the scope of the invention.

Each feature disclosed in the description, and (where appropriate) the drawings may be provided independently or in any appropriate combination.

# 2 Summary of Inventions

## 2.1 Hardware Implementation

| *No.* | *Description* | *Author* | *Date* |
|---|---|---|---|
| 1 | The SIF (background debug interface) instruction is all-1's in 32-bit form, and all-0's in 16-bit form. This allows, for example, external SIF control if the data bus is stuck at all-1's or all-0's | | |
| 2 | Placing the instruction size bit in the lowest bit position on a little endian architecture (or highest bit for big-endian) ensures that its location is independent of instruction size. | | |

## 2.2 Programmer's Model

| *No.* | *Description* | *Author* | *Date* |
|---|---|---|---|
| | | | |

## 2.3 Instruction Set

| *No.* | *Description* | *Author* | *Date* |
|---|---|---|---|
| 1 | The XAP3 instruction set not regular, but tailored to compiler behaviour, calling convention, etc. This is somewhat related to, say, the ARM Thumb processor, where the 16-bit Thumb instructions can only access the bottom eight registers. | | |
| 2 | Each instruction can be 16- or 32-bit, freely mixed on an instruction-by-instruction basis. This is a different approach to that of, say the ARM or MIPS16e where the processor operates in either 32-bit instruction or 16-bit instruction modes (typically on a function-by-function basis). | | |

| | | | |
|---|---|---|---|
| | 16-bit instruction decode done serially (16->32) as a decompression stage prior to full 32-bit decode. | | |
| 3 | The XAP3 has several compound (multi-action) instructions:<br>1. `ldm/stm` – load and store multiple registers in a single instruction (cf. ARM LDM/STM instructions). In addition, both instructions have reverse-order (".x") modes, and explicit user-mode register versions (".u") for use in operating systems.<br>2. The `movm` instruction can move up to five registers or small immediate values into R1-R5, saving up to five `mov` instructions. The order of writing to registers is from R1 to R5 (or vice-versa for the ".x" variant), allowing values to ripple through the registers.<br>3. `push` and `push.r` write multiple registers to the stack, and update the Stack Pointer register accordingly. In addition, and a feature unique to the XAP3, is the `push.r` instruction, which only pushes registers from R6 upwards, and then copies any of R1-R5 to matching registers from R6 onwards. This potentially saves many instructions at the beginning of a function for saving callee-save registers on the stack and copying argument registers into the upper registers.<br>4. Both `pop` and `pop.ret` load registers from memory, via the stack pointer, and update the Stack Pointer. In addition, and a feature we believe unique to the XAP3, is that both instructions take an additional argument (register or small immediate) and store that argument in R1, and update the condition flags as if that value was compared with 0. Additionally, the `pop.ret` instruction then executes a return via the Link Register. | | |
| 4 | Both `movm` source and `pop.ret` return value fields can be registers or immediates, encoded as a 5-bit field: one bit specifies register/immediate, the remaining four bits specifying either one of R0-R15, or an immediate value of –1, 1, …, 15. | | |
| 5 | Having R0 = 0 (cf. SPARC) allows #0 to mean other things for decode. | | |

| | | | |
|---|---|---|---|
| 6 | 17-bit signed immediates allow both 16-bit signed (-32768...+32767) and unsigned (0...65535) value ranges to be encoded in a single immediate.<br>The 4n+1 format is particularly noteworthy.<br>As is the 13s field which supports 3 regs + carry bit. | | |
| 7 | 16-bit instruction set is a true subset of the 32-bit instruction set: each 16-bit instruction can be mapped onto a single 32-bit instruction. | | |
| 8 | Uneven register visibility in 16-bit instructions to optimize for code density (cf ARM Thumb), although we have further optimized the design of the 16-bit instruction set to be even closer to the output of the compiler; for example, some 16-bit instructions allow access to R1-R4 and R6-R9, skipping R5 due to register usage behaviour of the compiler. | | |
| 9 | The SIF sleep instruction allows external debugger access to the processor registers and memory while the processor is in the low-power sleep state. | | |
| 10 | Support of position-independent code (PIC) requires computing addresses from some defined reference point. The XAP3 uses two reference points: the Program Counter (PC) for code and constant data references, and the Global Pointer (GP) for variable data references.<br>Whereas most architectures of this form provide rich addressing modes for accommodate these two (or more) reference points, we simplify the XAP3 instruction set with just two instructions for computing absolute addresses:<br>1. `mov.p` for PC-relative address generation<br>2. `mov.g` for GP-relative address generation | | |
| 11 | A lot of expressions in C compute some value dependent on the result of some test. We have developed the `sCC.*` instructions (where "CC" indicates a given condition code) to support this common code pattern, with two variants. The `sCC` instruction assigns to the output register either one of two registers ("`.r`" form) or one of a register and an immediate value ("`.i`" form), based on the state of the processor's condition code register.<br>Both forms are superior to the ARM's predicated | | |

| | | | |
|---|---|---|---|
| | movCC instructions as they combine both True and False choices into a single instruction. | | |
| 12 | The high-immediate ".h" instructions combined with 17-bit immediate support setting the upper half of a 32-bit value to a 16-bit value together with setting the lower sixteen bits to 0 or 1. This is particularly useful in creating bit masks where the remaining bits of a 32-bti mask must be 0 or 1. Our ".h" instructions combine two instructions (setting the upper and lower halves of a 32-bit value) into a single instruction. | | |
| 13 | Some instructions have a separate prime opcode for each register (e.g. mov.p). This allows coding long immediates in single instructions. | | |

## 2.4 Tool Support (assembler, compiler, linker, libraries)

| *No.* | *Description* | *Author* | *Date* |
|---|---|---|---|
| 1 | Both the processor and the toolchain have been developed together, and heavily optimized for code size. | | |
| 2 | Our assembly language mnemonic format uses a dot-notation to aid readability, of the form<br>base.parameter.type.size<br>or some combination of the above. | | |

## 2.5 Debug Support

| *No.* | *Description* | *Author* | *Date* |
|---|---|---|---|
| 1 | The XAP3 code has debug registers with a count conditional to trigger a breakpoint after some number of executions of the target instruction. This simplifies debugging of, say, loops.<br>As this is part of core, and not an external module, facilitates common debug environment both for development and in-field debugging. Especially for debugging user mode code by an operating system. | | |

| | | | |
|---|---|---|---|
| 2 | The hardware breakpoint registers in the XAP3 core supports software-implemented multi-stage breakpoint, using in-core debug registers, a debug module within OS, and textual or graphical interface to specify the breakpoint state machine. | | |

Cambridge Consultants

# XAP3 Specification

Prepared by
AGM, KLS, NEJ

Reviewed by
MDC

Secretary
CTS

**Cambridge Consultants Ltd**
Science Park
Milton Road
Cambridge
England CB4 0DW

Tel +44 (0)1223 420024
Fax +44 (0)1223 423373
info@CambridgeConsultants.com
www.CambridgeConsultants.com

C7245-S-001 v3.0.nc
12 May 2005

## Distribution

### Cambridge Consultants Ltd – Project Team

Project Reviewer

Project Manager

Technical Contributors

Project Secretary

# Table of Contents

**1 Introduction** 8
1.1 Error! Bookmark not defined.
1.2 Related Documents 10

**2 xIDE for XAP3** 11
2.1 xIDE Integrated Development Environment 11
2.2 Software Simulator 11
2.3 Hardware Emulator 11
2.4 Compiler, Assembler, Linker 11
2.5 Assembler Syntax 13
2.5.1 Syntax Strategy 13
2.5.2 Syntax Rules 13
2.5.2.1 General 13
2.5.2.2 Names 14
2.5.2.3 Directives and Labels 14
2.5.2.4 Numbers 14
2.5.2.5 Instruction Format 15
2.5.2.6 Instruction Mnemonics 15
2.5.2.7 Registers 16
2.5.2.8 Immediates 17
2.5.2.9 Addresses 18
2.5.3 Syntax Examples 18

**3 Key Hardware Features** 19
3.1 CPU 19
3.2 Memory 20
3.3 Hardware Implementation 20
3.4 Debug Support 20

**4 Processor Architecture** 21

**5 Programmer's Model** 23
5.1 Processor Mode 24
5.2 Processor State 25
5.3 Program State 25
5.4 Normal Registers 26
5.5 Breakpoint Registers 27
5.6 Special Registers 29
5.7 Flags 30
5.8 Memory Model 30
5.9 Reset State 31
5.10 Calling Convention 31
5.11 Application Models 32
5.11.1 Statically Linked 32
5.11.2 Dynamically Loaded 32

**6 Interrupts and Exceptions** 34
6.1 Overview 34
6.2 Exception Processing 35
6.3 Interrupt Processing 36

6.4 Returning from Interrupts and Exceptions 37
6.5 Nested Interrupts 37
6.6 Interrupt Vector Table 38
6.7 Exceptions 39
6.7.1 Reset 39
6.7.2 NMI 39
6.7.3 Trap 39
6.7.4 SingleStep 40
6.7.5 Break 40
6.7.6 UnknownInstruction 40
6.7.7 DivideByZero 41
6.7.8 NullPointer 41
6.7.9 IllegalInstruction 42
6.7.10 IllegalInstructionAccess and IllegalDataAccess 42
6.7.11 IllegalCodeAddress 43

**7 Addressing Modes** **45**
7.1 Register, Register, Immediate 45
7.2 Register, Register, Register 45
7.3 Load, Store 46
7.3.1 Displacement Addressing 46
7.3.2 Indexed Addressing 46

**8 Instruction Set** **47**
8.1 Notation Conventions 47
8.2 Structure of 32-bit Instructions 48
8.3 Structure of 16-bit Instructions 49
8.4 Base Instructions 50
8.5 Aliased Instructions 55
8.6 Flag Usage Conventions 56
8.7 Interpretation of Immediates 56
8.7.1 Low Immediates 56
8.7.2 High Immediates 56
8.7.3 Addresses 57
8.7.4 Register or Small Immediate Values 57
8.8 32-bit Instruction Opcodes 58
8.8.1 Opcode – OP[5:0] 58
8.8.2 Extended Opcode – OPA[3:0] 62
8.8.3 Extended Opcode - OPB[3.0] 63
8.8.4 Extended Opcode – OPC13[3:0] 64
8.8.5 Extended Opcode – OPC14[3:0] 65
8.8.6 Extended Opcode – OPC[3:0] 66
8.8.7 Extended Opcode – OPD14[3:0] 67
8.8.8 Extended Opcode – OPD[3.0] 68
8.8.9 Extended Opcode – OPE[3:0] 69
8.8.10 Extended Opcode – OPF[3:0] 70
8.9 16-bit Instruction Opcodes 71
8.9.1 16-bit Opcode – OP16[2:0] 71
8.9.2 16-bit Extended Opcode – OP16A0[3:0] 72
8.9.3 16-bit Extended Opcode – OP16B0 [3:0] 73
8.9.4 16-bit Extended Opcode – OP16A1[1:0] 73
8.9.5 16-Bit Extended Opcode – OP16A2[1:0] 74
8.9.6 16-bit Extended Opcode – OP16A5 74

8.9.7 16-bit Extended Opcode – OP16A6[1:0] 74
8.9.8 16-bit Extended Opcode – OP16A7 75
8.9.9 16-bit Extended Opcode – OP16B7 75
8.10 32-Bit Instruction Formats 76
8.10.1 Unconditional Branch Instructions 76
8.10.1.1 Absolute 76
8.10.1.2 PC Relative 76
8.10.1.3 Register 77
8.10.2 Conditional Branch Instructions 78
8.10.3 Conditional Set 79
8.10.4 Case Branch Instructions 80
8.10.5 Load and Store Instructions 81
8.10.5.1 Displacement Addressing (offset, Ra) – ld.*.i, st.*.i 81
8.10.5.2 Indexed Addressing (Rx, Ra) – ld.*.r, st.*.r, swap.r 82
8.10.5.2 1 swap.r Instruction 83
8.10.6 Load and Store Multiple Instructions 84
8.10.6.1 Modes 85
8 10.6.2 ldm.* 86
8 10.6.3 stm.* 87
8.10.7 push *, pop.* Instructions 88
8.10 7.1 push, push.r 89
8.10.7.1.1 push 89
8.10.7 1.1.1 push example 1 89
8.10.7.1.1.2 push example 2 90
8.10.7.1 2 push r 91
8.10.7.1.2.1 push.r example 92
8 10.7.2 pop, pop.ret 93
8.10.7.2.1 pop 93
8.10 7.2.1.1 pop example 1 94
8.10.7.2.1.2 pop example 2 94
8.10.7.2.2 pop.ret 95
8.10.7.2.2.1 pop.ret example 1 95
8.10.7.2.2.2 pop.ret example 2 96
8.10.8 mov Instructions 97
8 10.8.1 mov.p – PC-relative Offset 97
8.10.8.2 mov.g – GP-relative Offset 97
8.10.8.3 mov.i - Low Immediate 98
8.10.8.4 mov.h - High Immediate 98
8.10 8.5 mov.r - Register 99
8.10 8.6 movm – Multiple Register 99
8.10.9 ALU Instructions 101
8.10.9.1 Rules for Divide and Remainder 101
8.10.9.2 Reg-Reg-Immediate 102
8.10.9.3 Reg-Reg-Reg 103
8.10.10 Compare Instructions 104
8.10.10.1 Compare Assembler Format 104
8.10.10.2 Compare Immediate 105
8.10.10 3 Compare Register 106
8.10.11 Shift and Rotate Instructions 108
8 10.11 1 Reg-Reg-Immediate 108
8 10.11.2 Reg-Reg-Reg 108
8.10.11 3 rlinm Instruction 108
8 10.12 Special Register Instructions 110

8.10.13 DSP Instructions 111
8.10.13.1 flip.*.r Instruction 111
8.10.13.2 abs.r 112
8.10.13.3 msbit.r 112
8.10.13.4 acc.64.r 113
8.10.13.5 64-bit shift instructions 113
8.10 13.6 mult.sh.r 114
8 10.14 System Instructions 115
8.11 16-bit Instruction Formats 117
8.11.1 OP16A0 Register-Register Instructions 117
8.11.2 OP16B0 Instructions 119
8.11.3 OP16A1 Register-Immediate Instructions 119
8.11 4 OP16A2 16 and 8 bit ld and st Instructions for 16 and 8 bit Data 120
8.11.5 OP16A3 32-bit ld.i Instructions 121
8.11 6 OP16A4 32-bit st.i Instructions 122
8.11.7 OP16A5 movm and bra.p Instructions 122
8.11.7.1 movm 122
8.11.7.2 bra.p 123
8.11.8 OP16A6 Conditional Branch Instructions 124
8.11.9 OP16A7 Instructions 124
8.11.9.1 OP16A7 rlinm Instruction 124
8 11.9.2 OP16B7 ldm and stm Instructions 125
8.11.9.3 OP16B7 push/push.r Instruction 125
8.11.9.4 OP16B7 pop and pop.ret Instructions 126

**9 SIF Serial InterFace** **128**
9 1 SIF ASIC Block Diagram 128
9 2 SIF ASIC Modes 129
9.2.1 SIF Modes 129
9.2.1 1 SIF Mode = Normal 129
9.2.1.2 SIF Mode = Command 129
9.2.2 XAP3 SIF Processor Modes 130
9.2.2.1 XAP3 SIF Processor Mode = Normal 130
9.2.2.2 XAP3 SIF Processor Mode = Debug 131
9.3 When are SIF Operations executed ? 132
9.3.1 SIF Mode = Command 132
9.3.2 SIF Mode = Normal 132
9.3.2.1 Immediate SIF Handshake 133
9.3.2.2 Delayed SIF Handshake 133
9.3.2.3 Incomplete SIF Handshake (ending in SIF Lockup) 133
9.3.2.4 SIF Cancel (to clear SIF Lockup) 134
9.4 SIF Shift Register 135
9.4.1 Address Field 135
9.4.2 Control Field 135
9.4.3 Data Field 136
9.4.4 Status Field 136
9.4.5 Shift Register Bit Definitions 137
9.5 SIF Instructions 139
9 5.1 SIF Instruction Types 139
9.5.2 SIF Instruction Table 140
9.5.3 XAP3 Status Register 146
9.6 SIF Error Processing 147
9.6 1 SIF Parity Bits 147

9 6.1.1 PARITY_AC 147
9.6.1.2 PARITY_D 147
9.6.1.3 PARITY_S 147
9.6.2 SIF Error Reporting in Status Field 148
9.6.2.1 SIF Error Codes 148
9.6.2.2 Processor Error Codes 148
9.6.3 SIF Counters 149
9.7 SIF Resets 150

# 1 Introduction

This document is the specification for the XAP3, CCL's new 32-bit ASIC processor.

The associated software toolkit is 'xIDE for XAP3'. xIDE is CCL's generic Integrated Development Environment and runs on most computing platforms.

The XAP3 instruction set and toolkit have been developed before the hardware. The instruction set has been optimised to give good code density, leading to low power consumption and low silicon cost. The compile chain is based on the NCC C compiler (xap3ncc from Codemist) and GNU binutils for the assembler and linker.

XAP3 is aimed at 0.18, 0.13 and 0.09 µm CMOS processes. Initially, it is expected to run at clock speeds of 100MHz, giving 100MIPS. It is expected to contain about 50k gates with a typical memory manager.

XAP3 is intended to be used with on-chip RAM (normally 32-bit synchronous) and with off-chip RAM and Flash (normally 16-bit asynchronous). The default xap3_mmu supports 16 and 32-bit memories.

XAP3 has 16-bit and 32-bit instructions. These can be freely mixed in any pattern. The 16-bit instructions are a subset of the 32-bit instruction set. i.e all instructions are 16-bit aligned. This means that all 16-bit instructions are aligned. 32-bit instructions can be unaligned (to any even address).

XAP3 supports 4 operating modes (User, Supervisor, Interrupt, NMI). XAP3 has a load-store architecture. XAP3 includes powerful debug features including the SIF serial interface. The XAP3 SIF supports up to 8 on-chip processors and parity-checking.

XAP3 has three banks of registers. Normal Registers banks contains 16 general purpose 32-bit registers. Some of these are shadowed in Supervisor, Interrupt and NMI modes. The Breakpoint Registers bank supports 8 breakpoints, which are accessible from Supervisor, Interrupt and NMI modes. The Special Registers bank allows access to internal processor registers including the shadow register and flags.

XAP3 can be used with up to 4GB of memory. This is accessed with byte addresses (i.e. 32-bit address) and assumes Little Endian byte ordering for all words longer than 8-bits. All data memory is unaligned (8, 16 and 32-bit data can have any byte address).

Each program can access:

- 32MB of code/constants (mov.p limit)
- 32MB of global variables (mov.g limit)
- Up to 4GB of heap
- Up to 4GB of stack (limited to 64kB per function by xap3ncc which uses ld.i)
- Up to 2MB per function (conditional branch limit)

Programs are compiled to be relocatable by using PC-relative addressing for code/constants and GP-relative addressing for global variables. Programs can be loaded immediately next to each other in memory. There are no memory segments.

## 1.1

| | | |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

## 1.2 Related Documents

| *Reference* | *Title* | *Author* |
|---|---|---|
| | | |
| | | |
| | | |
| | | |

# 2 xIDE for XAP3

## 2.1 xIDE Integrated Development Environment

The software Integrated Development Environment (IDE) for XAP3 is based on CCL's xIDE. xIDE allows plugins to be created that add support for new processors, build systems and debuggers.

xIDE is based on Qt from Trolltech and can be compiled for Windows PC, Apple Mac OS X, and X11 platforms (including Sun and Linux).

The following documents describe xIDE for XAP3:

| | |
|---|---|
| C7066-TM-001 | xIDE User Manual |
| C7066-TM-003 | xIDE Python Object Model |
| C7066-TM-006 | XAP3 Plugin |

## 2.2 Software Simulator

The software simulator is part of the XAP3 plugin for the xIDE.

The Simulator allows Python models of hardware to be added to simulate the effect of hardware registers and interrupts.

## 2.3 Hardware Emulator

The Emulator is part of the XAP3 plugin for xIDE and provides a connection to the target board via a SIF physical interface. Cambridge Consultants sells PC test hardware and software drivers for the SIF. Other physical interfaces to the XAP3 Emulator (e.g SPI, I2C, JTAG, USB, RS232) can be developed if needed. The default target emulator board will be xEMU, Cambridge Consultants standard Digital Emulator (C6707-DL-002).

## 2.4 Compiler, Assembler, Linker

The first compiler will be based on Norcroft NCC (xap3ncc), with the intention of porting GCC later. Using GCC will allow us to have access to C++ in the future.

The assembler and linker will be derived from GNU binutils.

GCC and Binutils are released under the GNU General Public License. Binutils allows us to take advantage of standard ELF object files with embedded DWARF or Stabs debug information.

The word sizes that the C compiler should assume are as follows :

| *C compiler Type* | *No of bits* |
|---|---|
| Char | 8 |
| Short | 16 |
| Long | 32 |
| Int | 32 |
| Pointer | 32 |

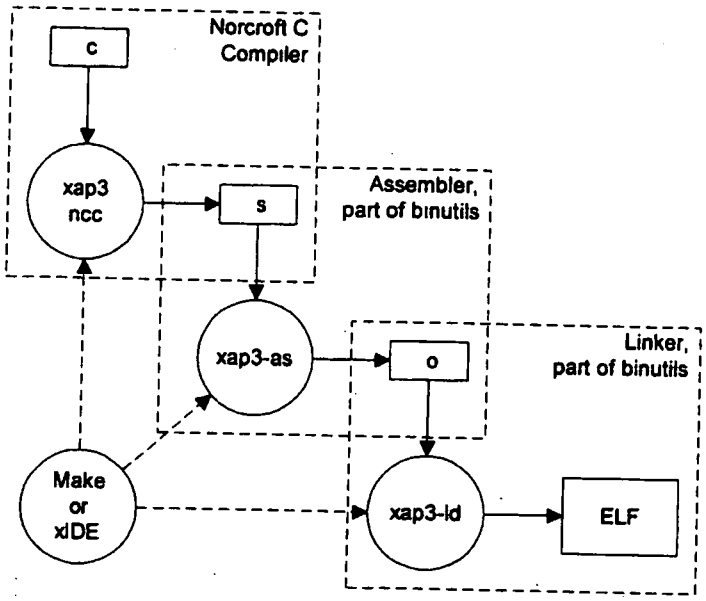
There is no hardware support for the data types; long long, float, double.

There is xap3ncc/library software support for float and double. These are treated identically as 64-bit words. They are stored in memory as structures and are unaligned (can have any byte address). The format used to store floats and doubles is a property of the compiler and library, not of the processor itself. The format used by the compiler and library for floating point variables will be IEEE 754 Standard, implemented as little-endian (bit order within words and word order within the structure). This is consistent with the hardware support for 8-bit and 16-bit variables in the processor itself.

Addresses are computed in two ways: PC-relative and GP-relative. The compiler will generate PC-relative addresses for all code and constant data objects, and GP-relative addresses for all other variables (global and static variables).

The instruction set contains 16-bit and 32-bit branch instructions. The compiler generates instruction mnemonics (in the .s file) and the assembler decides whether the instructions will be a 16 or 32-bit. All branch mnemonics generated by the compiler are PC relative branches. Conditional branches generated by the compiler will be to locations within the same function; the assembler will compact most of these branches to 16-bit instructions. Unconditional branches within a source file (module) will be either 16 or 32-bit instructions (decided by the assembler). Unconditional branches between modules will be 32-bit instructions and will be resolved by the linker. By having all inter-module branches as 32-bit instructions, we avoid the need to perform relaxation in the linker, which makes the process more efficient. Future linkers may perform relaxation, so that some unconditional branches between modules could be implemented as 16-bit instructions.

The diagram below shows the relationship between xap3ncc and Binutils.




## 2.5 Assembler Syntax

### *2.5.1 Syntax Strategy*

The choice of syntax rules for the assembler language used in .s files is driven by the following rules (most significant first) :

- No name-space pollution (e.g. for source files written in C)
- Occam's Razor – provide only one way to do things
- Binutils – follow the conventions normally used
- Case-sensitive – all reserved words will be lower-case
- C++ style syntax wherever possible (e.g. comments and numbers)
- XAP2 assembler – follow the conventions used there

### *2.5.2 Syntax Rules*

#### 2.5.2.1 General

The assembler will be line-based. Instructions are terminated by end-of-line.

Spaces and tabs are treated as white-space that delimit tokens.

All assembler will be interpreted as case-sensitive, except hex numbers.

All reserved words are lower-case (instruction mnemonics, registers, number prefixes ..)

Line Comments are from //-to end-of-line.

Block comments (can be across 1 or more lines) are from /* to */.

#### 2.5.2.2 Names

Names from C source will not be modified. Thus labels and variable names will be exactly the same as in the original C. However, local static symbols will have an assembler name of the format :

- basefilename.symbolname // e.g module.function

Hardware names will be prefixed by %. Normal registers will be referred to as %r0 to %r15.

The Special Registers and Breakpoint Registers will be referred to by the names defined in sections 5.6 and 5.5.

#### 2.5.2.3 Directives and Labels

Directives start with period (".") as the first non-blank character of the line. For example:

- `.org`
- `.file`
- `.text`

Labels start from the beginning of a line. They are appended with colon (":"). For example:

- `label_35A:`

#### 2.5.2.4 Numbers

Number formats used in XAP3 Assembler are :

| *Prefix* | *Example* | *Description* | *High bits up to Bit 31* | *Comment* |
|---|---|---|---|---|
| 0x | 0x12ab<br>0x12AB | Hex | Set to 0 | |
| | 103 | Signed Decimal | Sign Extended | Default |
| | -7521 | Signed Decimal | Sign Extended | Default |
| | 58297 | Unsigned Decimal | Set to 0 | |
| 0b | 0b11001 | Binary | Set to 0 | |

Positive signed decimal and unsigned decimal numbers look the same. They will be interpreted as required for that particular instruction (which may expect a signed or

unsigned number). The Assembler will reject numbers that are of invalid type or are out of range for that particular instruction. More detail in the later section on Immediates.

#### 2.5.2.5 Instruction Format

Instructions can take several forms :

- mnemonic label // Conditional and Unconditional Branches
- mnemonic %registers
- mnemonic %registers, #immediates // High and Low Immediates
- mnemonic %registers, @address

The convention is to have a tab between the mnemonic and the first argument. This should not be required by the binutils assembler, but should be generated by the C compiler.

Instructions must explicitly list every argument for each instruction. e.g. :

- ld.i %r2, @(1256, %r0) // NOT ld.i %r2, @1256

#### 2.5.2.6 Instruction Mnemonics

The mnemonic is a string in one of the following formats :

- Base
- Base.Parameters
- Base.Type
- Base.Parameters.Type
- Base.Parameters.Type.Size

Examples of Base are :

- add
- mov
- ld
- bra

Examples of Parameter are :

- n // not
- c // with carry
- x // exchange order (= reverse order)
- b // byte (swap bytes)
- z // zero-extend

- s // signed or sign-extend
- u // unsigned or user mode
- 64 // 64-bit operation
- 16 // 16-bit operation
- 8 // 8-bit operation

Examples of Type are :

- r // register
- i // low immediate from bit-0 upwards
- h // high immediate from bit-31 downwards
- a // absolute address
- p // PC-relative address
- g // GP-relative address

Examples of Size are :

- 2 // 2byte, 16-bit instruction
- 4 // 4byte, 32-bit instruction

Instruction mnemonics do not normally include Size. It is only used when the xap3ncc compiler wants to force the binutils assembler/linker to use a particular instruction size. This is sometimes needed for branch instructions.

Here are some alternative versions of the add instruction :

- add.r // Add Rt to Rs and store result in Rd
- add.c.r // Add Rt to Rs with carry and store result in Rd
- add.i // Add Low Immediate to Rs and store result in Rd
- add.h // Add High Immediate to Rs and store result in Rd

#### 2.5.2.7 Registers

There can be several Registers in each instruction.

They will be in the following sequence from left to right :

- Rd // Destination Register
- Rs // Primary Source Register
- Rt // Secondary Source Register
- Rx // Index Address Register (scaled by 4, 2 or 1 for 32, 16 and 8 bit data)
- Ra // Base Address Register (address in bytes)

Each Register will be separated by, and prefixed by %.

Some instructions (ldm.*, stm.*, push, pop.*) require a list of registers as an argument. The number of registers can vary. Such lists are enclosed in {}. e.g. :

- ldm {%r1, %r5, %r8, %r11}, @(16, %r12) // Load 4 regs from memory

Within such register lists, registers can be specified in ranges as follows :

- %r1-%r8 // The first register must be lower than the second.

This enables more compact Assembler code (from the C compiler and Disassembler).

Within the register list, the same register must not be specified more than once :

- {%r1-%r3, %r2} // Not valid because %r2 has been specified twice
- {%r3, %r6-%r8, %r10-%r13} // This is a valid register list

#### 2.5.2.8 Immediates

There can be several Immediates in each instruction. Each Immediate will be separated by, and prefixed by #. For example:

```
rlinm.s %r4, %r6, #0, #15, #31
```

Immediate values in the assembly files are treated as 32-bit numbers. It is the responsibility of the assembler to decide whether the immediate value is compatible with the values permitted by the instruction mnemonics.

The assembler will sign-extend positive and negative decimal numbers, but not hex or binary. The assembler will then calculate whether the resultant 32-bit number is in a valid range for the Immediate field for that particular instruction. e.g :

```
add.i   %r1, %r2, #0x1234       // valid positive immediate
add.i   %r1, %r2, #35           // valid positive immediate
add.i   %r1, %r2, #-356         // valid negative immediate
add.i   %r1, %r2, #0xFFFF1234   // valid negative immediate
add.i   %r1, %r2, #0x111234     // invalid (immediate > 17 bit)
add.i   %r1, %r2, #0xABCD1234   // invalid (immediate > 17 bit)
add.i   %r1, %r2, #-100000      // invalid (immediate > 17 bit)

add.h   %r1, %r2, #0x12340000   // valid
add.h   %r1, %r2, #0x1234FFFF   // valid
add.h   %r1, %r2, #0x1234ABCD   // invalid
add.h   %r1, %r2, #0x1234       // invalid

mov.h %r1, #0x12340000          // valid
mov.h %r1, #0x1234FFFF          // valid
mov.h %r1, #0x12345678          // invalid
mov.h %r1, #0x1234              // invalid
```

In practice it will always be easier (and often essential) to specify the high immediate in a .h instruction as a hex number.

#### 2.5.2.9 Addresses

There can only be one Address in each instruction. This will be prefixed by @.

Address indirection is specified as one of :

- @(offset, %Ra) @(-56, %r15) // Ra = byte address
- @(%Rx, %Ra) @(%r4, %r12) // Rx = scaled addr, Ra = byte addr

### 2.5.3 *Syntax Examples*

Here are some valid instructions in XAP3 Assembler :

```
ld.i    %r2, @(1256, %r0)   // Load value at absolute address 1256 into Register2
st.r    %r8, @(%r3, %r11)   // Store register R8 at specified address
add.r   %r4, %r3, %r9       // Add R9 to R3 and store in R4
add.c.r %r7, %r11, %r4      /* Start of Block comment
foo bar
        End of Block Comment */   add.r   %r6, %r3, %r9 // this add.r is executed
or.i    %r3, %r7, #-563             // OR –563 with R7 and store in R3
and.h   %r8, %r10, #0xABcd0000      // AND hex ABCD with R10 and store in R8
xor.h   %r13, %r5, #0x1230000       // XOR 0x1230000 with R5 and store in R13
stm     {%r1, %r5, %r8, %r11}, @(24, %r12)      // Store 4 regs to memory
```

# 3 Key Hardware Features

## 3.1 CPU

32-bit internal architecture.

Unified linear 32-bit address space (4GB).

Each program can use up to 32MB (code/constants on 2byte boundary) + 32 MB (global variables on 4byte boundary). These can be positioned anywhere in memory. Separate programs can be loaded immediately next to each other. There are no memory segments to observe.

Byte addressing. All memory accesses assume Little-Endian byte ordering. If a 32-bit word is at address 0x100, then the bits can be found at the following addresses :

- word[7:0] = address 0x100
- word[15:8] = address 0x101
- word[23:16] = address 0x102
- word[31:24] = address 0x103

32-bit and 16-bit instructions are aligned on 16-bit address boundaries. i.e 16-bit instructions are aligned. 32-bit instructions can be aligned or unaligned (but they always have an even address).

All data (8-bit, 16-bit, 32-bit, arrays and structures) is unaligned (can have any byte address). There are compiler options for packed structures (smaller, slower) and for unpacked structures (bigger, faster).

All addresses and pointers are stored in registers as full 32-bit byte addresses. All address comparisons are done on full 32-bit byte addresses. Even though an individual program operates within a smaller memory space (32MB for code + constants), the addresses stored and compared in registers are always full 32-bit byte addresses. They can therefore use the full 4GB that the XAP3 can access.

There are 3 banks of up to 16 registers. All registers are 32-bit :

- Normal Registers — see section 5.4
- Breakpoint Registers — see section 5.5
- Special Registers — see section 5.6

The only instruction that can update a Special Register is movr2s. This can only be used by the privileged Supervisor, Interrupt and NMI modes. Similarly the movr2b instruction allows the privileged modes to update the breakpoint registers. All other processing is performed on the Normal registers.

Aimed at fixed point 32-bit processing. Support for 16-bit, and 8-bit operations

Stack is 32 bits wide. Stack is word aligned (stack pointer address is always a multiple of 4). Stack Pointer address changes by ±4 (byte addressing). Stack grows downwards (standard for GCC and NCC).

Good code density, comparable to ARM9TDMI.

Good support for C compilation (suitable for GCC and NCC).

User, Supervisor, Interrupt and NMI Modes.

### 3.2 Memory

The XAP3 can be used with any combination of 8, 16 and 32-bit synchronous and asynchronous memories.

The typical configuration will be to have internal 32-bit synchronous memories (RAM and registers) and external 16-bit asynchronous memories (parallel RAM and parallel Flash).

### 3.3 Hardware Implementation

Clock speeds of at least 100MHz on 0.18 μm CMOS

Estimated size is 50k gates

Optimised for low cost and low power

### 3.4 Debug Support

The SIF is described in section 9.

SingleStep and BreakExceptions are described in sections 6.7.4 and 6.7.5.

The breakpoint registers are described in section 5.5.

# 4 Processor Architecture

There are two versions of the XAP3 processor :

| *Processor* | *Features* | *Documentation* |
|---|---|---|
| XAP3a | 2-stage pipeline<br>Smallest possible XAP3<br>Slower<br>Lower power | C7245-TM-001 |
| XAP3b | 4-stage pipeline<br>Larger<br>Faster<br>Higher power | C7245-TM-002 |

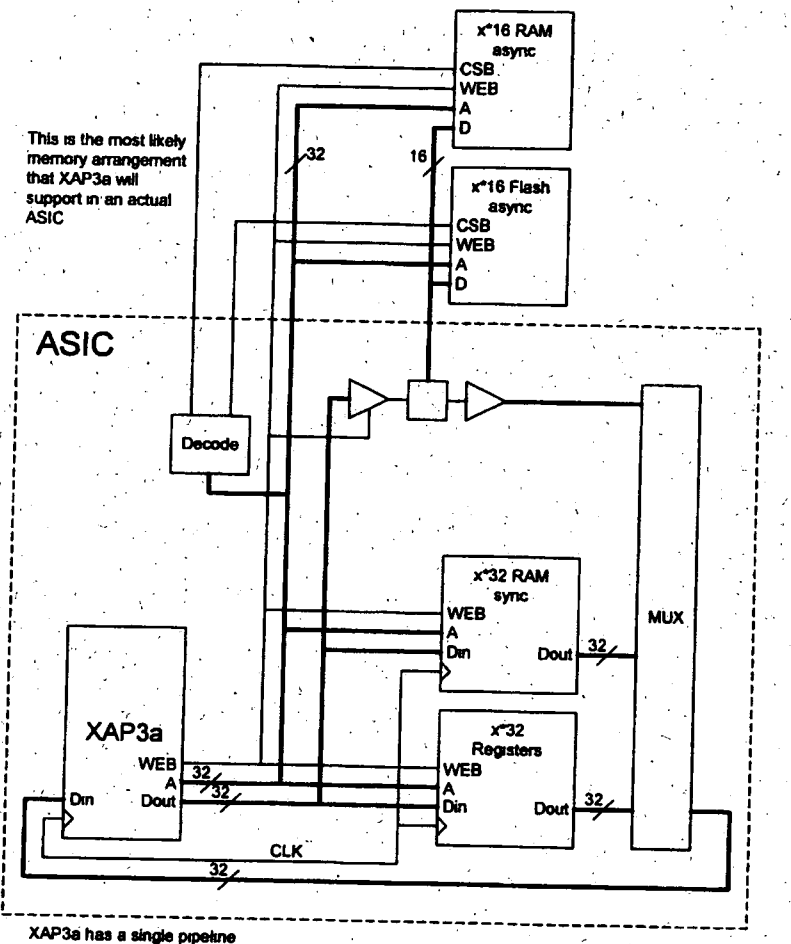
The default XAP3 architecture consists of:

- xap3_core - fixed and not changed from one ASIC to another
- xap3_debug – debug support to a serial interface (e.g SIF, SPI, $I^2C$, JTAG etc)
- xap3_mmu – Memory manager unit, likely to be changed from one ASIC to another
- xap3_ivc – Interrupt vector controller, likely to be changed from one ASIC to another

The debug serial interface normally used is SIF. This is a patented non-invasive serial interface developed by Cambridge Consultants. It is described in more detail at the end of this specification. The SIF is outside the XAP3. A single SIF can support up to 8 XAP3s (or a mixture of processors). Each processor can have its own address space (up to 4GB each), or can use shared address spaces.

Here is a block diagram for XAP3a, which is a minimum implementation of the XAP3 instruction set, using a single physical bus for all memory accesses (Prog and Data) :

XAP3a - Hardware Architecture




XAP3a has a single pipeline
It supports a single memory bus (32bit data (in & out), 32bit address) only (even XAP3a_CORE)
Instructions are implemented in the minimum number of clocks
RRR & RRI instructions will take 1 clock.
LD & ST instructions will take 2 clocks
This implies qute long combinational paths

There is more information on how a single ASIC can support multiple on-chip processors in the SIF section at the end of this specification.

# 5 Programmer's Model

| | User Mode Registers | Supervisor Mode Registers | Interrupt Mode Registers | NMI Mode Registers | Assembler Syntax |
|---|---|---|---|---|---|
| **Normal Registers** | 31 0 | 31 0 | 31 0 | 31 0 | |
| Constant Zero | R0 | | | | %r0 |
| Function Argument 1 and Return Value | R1_U | R1_S | R1_I | R1_N | %r1 |
| Function Argument 2 | R2_U | R2_S | R2_I | R2_I | %r2 |
| Function Argument 3 | R3 | | | | %r3 |
| Function Argument 4 | R4 | | | | %r4 |
| Function Argument 5 | R5 | | | | %r5 |
| | R6 | | | | %r6 |
| | R7 | | | | %r7 |
| Accumulator 0 | R8 | | | | %r8 |
| | R9 | | | | %r9 |
| Accumulator 1 | R10 | | | | %r10 |
| | R11 | | | | %r11 |
| | R12 | | | | %r12 |
| | R13 | | | | %r13 |
| Link Register (LR) | R14_U | R14_S | R14_I | R14_N | %r14 |
| Stack Pointer (SP) | R15_U | R15_S | R15_I | R15_I | %r15 |
| | 31 0 | | | | |
| Program Counter | PC | | | | |
| **Special Registers** | 31 0 | 31 0 | 31 0 | 31 0 | |
| Flags | FLAGS | | | | %flags |
| Saved Flags | | SFLAGS_S | SFLAGS_I | SFLAGS_N | %sflags_s/i/n |
| Global Pointer (GP) | GP_U | GP_S | GP_I | GP_I | %gp_u/s/i |
| Interrupt Vector Table Base Address | IVTB | | | | %ivtb |
| Breakpoint Enable | BRKE | | | | %brke |
| **Breakpoint Registers** | 31 0 | | | | |
| Breakpoint 0 | BRK0 | | | | %brk0 |
| Breakpoint 1 | BRK1 | | | | %brk1 |
| Breakpoint 2 | BRK2 | | | | %brk2 |
| Breakpoint 3 | BRK3 | | | | %brk3 |
| Breakpoint 4 | BRK4 | | | | %brk4 |
| Breakpoint 5 | BRK5 | | | | %brk5 |
| Breakpoint 6 | BRK6 | | | | %brk6 |
| Breakpoint 7 | BRK7 | | | | %brk7 |
| Breakpoint 4 Count | BRK4COUNT | | | | %brk4count |
| Breakpoint 5 Count | BRK5COUNT | | | | %brk5count |
| Breakpoint 6 Count | BRK6COUNT | | | | %brk6count |
| Breakpoint 7 Count | BRK7COUNT | | | | %brk7count |
| Breakpoint 6 Mask | BRK6MASK | | | | %brk6mask |
| Breakpoint 7 Mask | BRK7MASK | | | | %brk7mask |
| Breakpoint 6 Data | BRK6DATA | | | | %brk6data |
| Breakpoint 7 Data | BRK7DATA | | | | %brk7data |

**Register Encodings**

FLAGS
SFLAGS_S
SFLAGS_I
SFLAGS_N

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | E | T | B | M1 | M0 | V | C | N | Z |

BRKE

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | E7 | R7 | W7 | 0 | E6 | R6 | W6 | 0 | E5 | R5 | W5 | 0 | E4 | R4 | W4 | 0 | E3 | R3 | W3 | 0 | E2 | R2 | W2 | 0 | E1 | R1 | W1 | 0 | E0 | R0 | W0 |

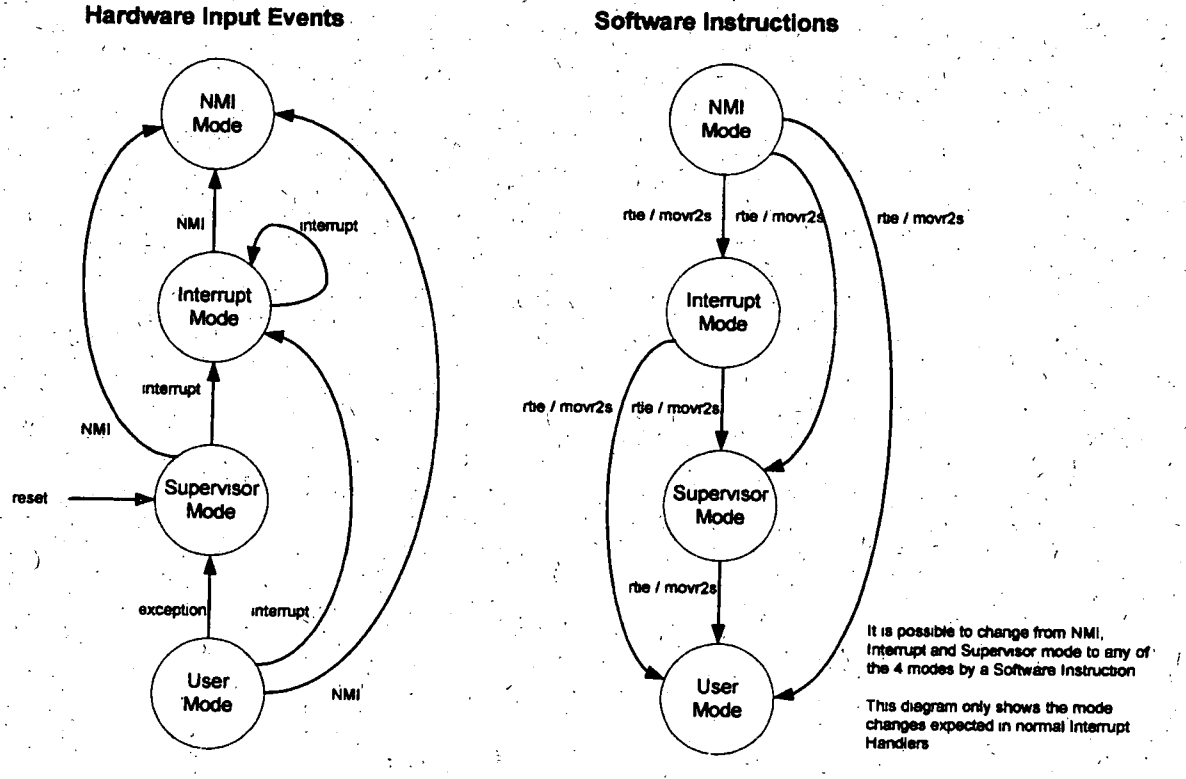
## 5.1 Processor Mode

There are four modes of operation:

- User
- Supervisor
- Interrupt
- NMI (Non-Maskable Interrupt)

User mode allows unknown software to be run safely without affecting the operation of the supervisor, interrupt and NMI mode code. The MODE[1:0] signal from xap3_core to xap3_mmu indicates whether the xap3_core is in User, Supervisor, Interrupt or NMI Mode. The xap3_core has a 'memory exception' input that allows the xap3_mmu to indicate when a memory access is attempted that is illegal in the current operating mode.

The diagram below shows how the mode changes will take place.




We expect the four modes to be used as follows:

- NMI Mode – Special interrupt mode for supporting non-maskable interrupts.
- Interrupt Mode – Hardware Drivers (IVC hardware generates defined interrupt addresses for specified events)

- Supervisor Mode – Operating System
- User Mode – Application Software

The movr2s instruction can be used to change from any privileged mode to any mode.

It is expected that the majority of code will execute in User Mode and operating system functions will be implemented in Supervisor Mode. User mode code can call Supervisor Mode code by using the trap instruction.

The processor resets to Supervisor mode. The initialisation code is then responsible for initialising the stack and starting User Mode. The mode switch is implemented using the `movr2s` instruction to write to the FLAGS register.

User mode can only access User registers. The Interrupt, NMI and Supervisor modes are known as privileged modes. They can access their own and the User Mode registers. Supervisor Mode cannot access Interrupt or NMI Mode registers and vice versa. In order to initialise Interrupt and NMI Mode registers, programmers must disable interrupts and manually switch to Interrupt or NMI Mode by writing to the FLAGS register.

Code executed in User Mode cannot directly write to the FLAGS register.

## 5.2 Processor State

The processor can be in two states:

- Sleeping
- Awake

When the processor is awake, SIF accesses can only take place when the CPU is executing a sif instruction.

There are two sleep modes. One that allow SIF accesses and one that doesn't.

## 5.3 Program State

The state of the program refers to whether the core is executing instructions or not. The program can be in three states:

- Running
- Stopped
- Stepping

These states will normally only be encountered during interactive debugging.

## 5.4 Normal Registers

Registers R1, R2, R14 and R15 are shadowed such that there is one of each of these registers per processor mode. For these registers, the actual register accessed by an instruction depends on the mode in which the processor is operating.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| %r0 | 0000 | R0 | Constant zero |
| %r1 | 0001 | R1_U or R1_S or R1_I or R1_N | Dependent on processor mode. Used as function argument 1 and return value. |
| %r2 | 0010 | R2_U or R2_S or R2_I | Dependent on processor mode. R2_I is used for NMI and Interrupt Mode. Used as function argument 2. |
| %r3 | 0011 | R3 | Used as function argument 3. |
| %r4 | 0100 | R4 | Used as function argument 4. |
| %r5 | 0101 | R5 | Used as function argument 5. |
| %r6 | 0110 | R6 | |
| %r7 | 0111 | R7 | |
| %r8 | 1000 | R8 | Accumulator 0 bits [31:0]. |
| %r9 | 1001 | R9 | Accumulator 0 bits [63:32]. |
| %r10 | 1010 | R10 | Accumulator 1 bits [31:0]. |
| %r11 | 1011 | R11 | Accumulator 1 bits [63:32]. |
| %r12 | 1100 | R12 | |
| %r13 | 1101 | R13 | |
| %r14 | 1110 | R14_U or R14_S or R14_I or R14_N | Dependent on processor mode. Used as Link Register (LR). |
| %r15 | 1111 | R15_U or R15_S or R15_I | Dependent on processor mode. R15_I is used for NMI and Interrupt Mode. Used as Stack Pointer (SP). |

The C compiler will use R15 as a Stack Pointer. R1, R2, R14 and R15 are shadowed in Supervisor and Interrupt modes. Only, R1 and R14 are shadowed in NMI mode. NMI shares the Interrupt mode R1 and R15 shadow registers.

At any one time, a register is only used for a single variable (whether it is 8, 16 or 32 bit). All arithmetic, compare and logical operations operate on full 32-bit words. Load

and Store instructions have 16-bit and 8-bit forms (sign and zero extended), however, allowing programs to operate on 16-bit and 8-bit variables.

## 5.5 Breakpoint Registers

Software development on XAP3 can implement break points 2 ways :

- Swap normal instruction with brk instruction in memory. This cannot support conditional breaks. This can only be used in memory that supports individual word writes. 'xIDE for XAP3' chooses to use these where possible, as they are an unlimited resource.
- Use one of the 8 break registers. These can support conditional breaks. These can be used with any kind of memory (because the memory itself is not modified).

The XAP3 contains eight breakpoint registers, BRK0-7, that can be configured to stop the processor when an address is read, written, or executed.

In addition to this, BRK4-7 have matching count registers, such that the breakpoint is triggered if the address condition described above matches and the value of the corresponding count register is zero. At reset the count registers contain zero, which means the breakpoint registers behave like simple breakpoints. When an address match happens and the count register is not zero, the count register is decremented by one. Note that the count value never cycles: once it reaches zero it stays there, until changed by a write with the `movr2b` instruction.

In addition to this, BRK6 and BRK7 have associated data and mask registers that allow the breakpoints to be conditional on the data read/written by an instruction. These will only trigger when the address matches, the count is zero and the data and mask conditions are satisfied. The data registers specify 1's and 0's to match against; the mask registers specify don't cares where 1 = match, 0 = don't care. For example, suppose we want to break on a data value where bit 12 is one, bit 11 is zero and we don't care about the rest of the bits. We specify:

```
%brk6data = 00000000000000000001000000000000
%brk6mask = 00000000000000000001100000000000
```

When a mask register is set to zero the data matching is effectively disabled because all data values will trigger a match. So the break will occur when the address matches and the count is zero.

The count and data match features can be used to implement sophisticated conditional breakpoints in the debugger.

Providing privileged modes access to the breakpoint registers allows Supervisor Mode debuggers to be implemented that allow User Mode tasks to be debugged, without needing to stop the processor. This allows, for example, the main operating system to remain running when debugging User Mode applications.

The bits in the BRKE register are described below.

| *Bit(s)* | *Flag(s)* | *Name* | *Description* |
|---|---|---|---|
| 0, 4, 8, 12, 16, 20, 24, 28 | W0, W1, W2, W3, W4, W5, W6, W7 | Write | If set, an instruction making a memory write to the address matching the corresponding BRKn register will halt the processor after the instruction has executed. The PC will point at the next instruction to be executed. |
| 1, 5, 9, 13, 17, 21, 25, 29 | R0, R1, R2, R3, R4, R5, R6, R7 | Read | Similar to above, but for memory reads. |
| 2, 6, 10, 14, 18, 22, 26, 30 | E0, E1, E2, E3, E4, R5, E6, E7 | Execute | If set, an instruction fetch from the address matching the corresponding BRKn register will halt the processor before the instruction is executed. The PC remains at the address of the instruction that caused the break. |

Note: the BRKE register is one of the Special Registers. The special registers are described in section 5.6.

In User mode, the above break conditions will also halt the processor if the B flag is 0. If the B flag is 1 the processor will not halt, but will generate a Break exception instead (see section 6.7.5).

The breakpoint registers can be written and read by privileged modes using the `movr2b` and `movb2r` instructions, see below. The `movs2r` and `movr2s` instructions allow access to the BRKE register.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| `%brk0` | 0000 | BRK0 | |
| `%brk1` | 0001 | BRK1 | |
| `%brk2` | 0010 | BRK2 | |
| `%brk3` | 0011 | BRK3 | |
| `%brk4` | 0100 | BRK4 | |
| `%brk5` | 0101 | BRK5 | |
| `%brk6` | 0110 | BRK6 | |
| `%brk7` | 0111 | BRK7 | |
| `%brk4count` | 1000 | BRK4COUNT | |
| `%brk5count` | 1001 | BRK5COUNT | |
| `%brk6count` | 1010 | BRK6COUNT | |
| `%brk7count` | 1011 | BRK7COUNT | |
| `%brk6mask` | 1100 | BRK6MASK | |
| `%brk7mask` | 1101 | BRK7MASK | |
| `%brk6data` | 1110 | BRK6DATA | |

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| `%brk7data` | 1111 | BRK7DATA | |

## 5.6 Special Registers

The `movs2r` and `movr2s` instructions allow Interrupt, Supervisor and NMI mode access to the Special Registers. These are shown in the table below.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| `%flags` | 0000 | FLAGS | |
| `%sflags_s` | 0001 | SFLAGS_S | Saved Flags. The processor copies FLAGS into this register when an exception happens. |
| `%sflags_i` | 0010 | SFLAGS_I | Saved Flags. The processor copies FLAGS into this register when an interrupt happens. |
| `%sflags_n` | 0011 | SFLAGS_N | Saved Flags. The processor copies FLAGS into this register when an NMI happens. |
| `%gp_u` | 0100 | GP_U | The Global Pointer used by `mov.g` in User Mode |
| `%gp_s` | 0101 | GP_S | The Global Pointer used by `mov.g` in Supervisor Mode |
| `%gp_i` | 0110 | GP_I | The Global Pointer used by `mov.g` in Interrupt and NMI Mode |
| `%ivtb` | 0111 | IVTB | |
| `%brke` | 1000 | BRKE | |
| | 1001 | *reserved* | |
| | 1010 | *reserved* | |
| | 1011 | *reserved* | |
| | 1100 | *reserved* | |
| | 1101 | *reserved* | |
| | 1110 | *reserved* | |
| | 1111 | *reserved* | |

The format of the Flags registers is descibed in section 5.7.

The format of the BRKE register is described in section 5.5.

## 5.7 Flags

The table below descirbes the layout of the FLAGS, SFLAGS_S, SFLAGS_I and SFLAGS_N registers.

| *Bit* | *Flag* | *Name* | *Description* |
|---|---|---|---|
| 8 | E | Interrupt Enable | If set, normal interrupts are enabled. (NMI cannot be disabled) |
| 7 | T | Single Step | If set, each User Mode instruction that is executed will throw the SingleStep exception. |
| 6 | B | Break | If set and the processor is in User Mode, a break as a result of a brk instruction or the BRK registers will throw the Break exception. |
| 5:4 | M[1:0] | Mode | 00: Supervisor Mode<br>01: User Mode<br>10: Interrupt Mode<br>11: NMI Mode |
| 3 | V | Overflow | Set if signed arithmetic overflowed |
| 2 | C | Carry | Set if unsigned arithmetic overflowed |
| 1 | N | Negative | Set if result is negative (Rd[31] = 1) |
| 0 | Z | Zero | Set if result is zero (Rd = 0) |

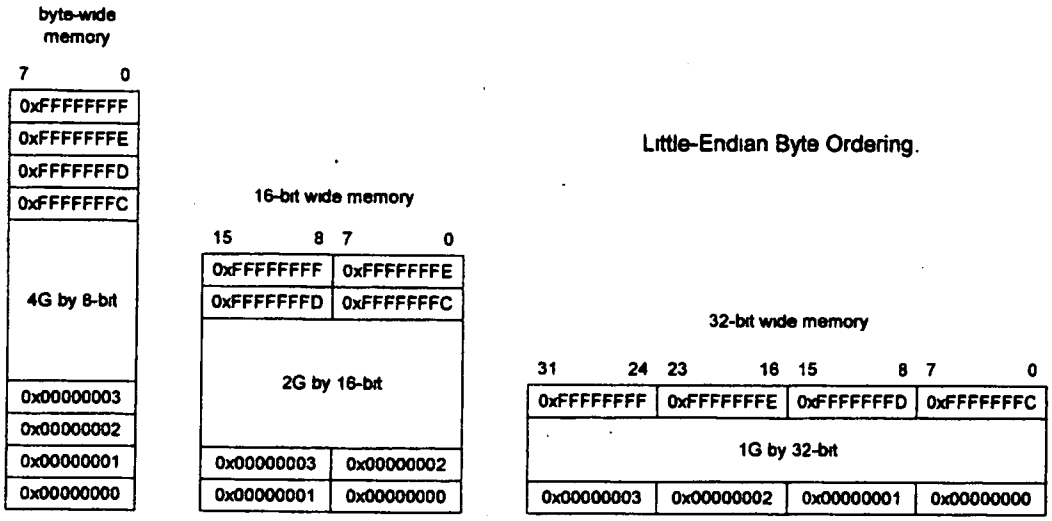
When in the Supervisor, NMI or Interrupt Modes, the entire FLAGS register can be modified by executing the movr2s instruction. The movs2r instruction allows the FLAGS register to be read. The `irqe` and `irqd` instructions can be executed in Supervisor, NMI and Interrupt Modes. They are used to enable and disable interrupts.

The mode bits of the FLAGS register are automatically changed when interrupts or exceptions happen. The other bits remain unchanged. At reset the processor will start executing code in supervisor mode.

The condition flags, Z, N, C, and V will be modified as instructions are executed. See the instructions description for details of the instructions that modify flags.

## 5.8 Memory Model

The memory model has a 32-bit address space (4G Bytes), containing both code and data. The address space is byte addressed. All pointers are kept as full 32-bit byte addresses, and will thus fit in a 32-bit register.




All instructions are aligned to a 16-bit boundary (i.e. bit 0 of the address is 0), but it is not necessary for 32-bit instructions to be aligned to a 32-bit boundary. This means that all 16-bit instructions are aligned, whereas 32-bit instructions can be unaligned.

All data objects are unaligned. This means that 8, 16 and 32-bit data can have any byte address. The diagram above only shows aligned data objects.

All words use Little Endian byte ordering, i.e. the low bits of 16-bit or 32-bit or 64-bit words are at the low byte address.

## 5.9 Reset State

All registers and flip-flops in XAP3 are asynchronously set or reset when the RESETB input pin goes low. Wherever possible, the registers and flip-flops should be reset and not set.

Memories (internal and external) cannot be reset and so will not be.

At reset, the XAP3 will be in the following state :

- Program Counter = 0 (i.e. code execution starts from address 0)
- Interrupts disabled
- In Supervisor mode
- Program Running
- Processor Awake
- SIF in Normal mode

## 5.10 Calling Convention

The bsr.p or bsr.a instruction is used to make function calls. The effect of bsr.p or bsr.a is to transfer the address of the next instruction to the link register, R14, and to transfer the destination address to the PC. Control is returned to the calling function when the contents of R14 is transferred to PC. In practice, the compiler generates the bsr.p instruction.

The calling convention for the XAP3 is :

- The called function does not need to preserve the contents of R1, R2, R3, R4, R5 and R14. R14 = Link Register.
- The called function is responsible for preserving the remaining registers (R6 – R13, R15) across function calls. This convention is implemented in the xap3ncc C compiler. It is not a hardware property.
- R15 (Stack Pointer) is used to operate a fully covered stack, so interrupt handlers don't need to allow for a partially covered stack as in XAP2 systems. A fully covered stack is when function data is always at positive or zero offset from the stack pointer. A partially covered stack allows negative offsets to be used.
- Arguments are passed to functions in R1, R2, R3, R4, R5 then on the stack. When a 16-bit value is passed to a function in a register, the whole register is used. The upper bits are sign or zero extended according to the type. Structures that are passed in registers are converted to a sequence of 32-bit words.
- The called function places the C function return value in R1 (convention implemented in xap3ncc).
- Special functions may store extra return data in R2, R3, R4, R5.
- When a temporary register is needed, xap3ncc uses R14 (LR)
- Code is compiled independent of processor mode.
- When the processor is interrupted, some assembler code must be executed to store the contents of R3-R5 before the C routine can be called. This is not needed for R1 or R2 as they have hardware shadow registers. C functions will not corrupt (R6-R13, R15), but may corrupt (R1-R5, R14).
- The flags will be set according to the result of comparing the return value, held in R1, with #0. For functions that have no return value (`void` functions) the flags are undefined.

## 5.11 Application Models

The XAP3 can support various application models.

### *5.11.1 Statically Linked*

This is the basic application model where the locations of all functions and data are known at link time.

### *5.11.2 Dynamically Loaded*

The XAP3 supports systems where applications can be loaded to memory locations that are not known at link time. Statically allocated variables are accessed relative to the GP register.

The relative positions of code and data can be varied at load-time. PC relative addressing for code and constants (probably stored in Flash) means only minimal load time fixups are needed. GP relative addressing for global variables (probably stored in RAM) means only minimal load time fixups for data, too.

If the same program is being run in more than one instance, then the code/constants only need to be loaded once. Each instance will have its own GP value, pointing to its own area of RAM.

Absolute addressing can be used to access code and data at fixed addresses. This means that no load time fixups are required to access absolute code and data locations within the system.

Fixups will be necessary when static initialisers contain an address of a static variable. The location of the static variable is not known until link time, hence the loader will need to fixup the initialisation constant. This code would require a load time fixup when used as part of a dynamically loaded application.

```
static int stVar;
static int * stPtr = &stVar; // Fixup needed because of &
```

# 6 Interrupts and Exceptions

This section describes the interrupt and exception handling mechanisms of XAP3. The objectives of the design are:

- to keep the XAP3 core as simple as possible, in keeping with the general design philosophy, and to minimise the burden of implementing simple controller applications.
- to provide a level of capability that will support a wide variety of operating systems, from simple round-robin schedulers, to modern multi-tasking, virtual-memory-enabled operating systems, such as Embedded Linux, vxWorks, Nucleus, etc.
- to support debuggers that allow users to interactively debug a User Mode application without halting an underlying operating system.

In the description below the following terms are used.

- **Handler** – a piece of code that handles the response to an interrupt or exception.
- **Exception** – a special event triggered by some action of the XAP3 core, e.g., dividing by zero, or undefined instruction.
- **Interrupt** – a special event triggered by some action external to the XAP3 core, e.g., a timer reaching its terminal count, or a serial port receiving a character.
- **Vector** – the mapping between an interrupt or exception identifier and the corresponding handler.

## 6.1 Overview

The XAP3 core has four operating modes:

- User Mode
- Supervisor Mode
- Interrupt Mode
- NMI Mode

Exceptions may be thrown by code executing in User Mode and cause a switch to Supervisor Mode. Exceptions cause interrupts to be disabled. The exception handler can re-enable them using the `irqe` (Interrupt Enable) instruction.

The XAP3 has two types of interrupt:

- Interrupts – can be enabled and disabled
- Non-Maskable Interrupts – cannot be disabled

Interrupts cause the processor to switch to Interrupt Mode and to disable further interrupts. Interrupts can be enabled and disabled using the `irqe` (Interrupt Enable) and `irqd` (Interrupt Disable) instructions.

Non-Maskable Interrupts force a switch to a special NMI Mode. Just enough information is preserved in the processor to allow return to the interrupted code. Since NMI Interrupts cannot be disabled, special care should be taken in writing NMI handlers and in the design of the hardware to ensure that the software can handle multiple NMIs (or simply disallow multiple NMIs).

Note that as NMI Mode is a special form of Interrupt Mode it also shares a single stack with Interrupt Mode (SP_I). Systems which implement NMI handlers should allow space on the Interrupt Mode stack for any NMI handlers that may be called during execution of Interrupt handlers.

Supervisor Mode is never re-entrant. Interrupt Mode can be re-entrant if interrupts are re-enabled within the Interrupt Mode handler.

## 6.2 Exception Processing

Exceptions in the XAP3 are precise exceptions, in that we can know the exact instruction that caused the exception. This allows the operating system to be very precise in its handling of the exception.

The cost of precise exceptions comes when implementing a pipelined processor core, since there can be several instructions in flight at the same time, and at each stage in the pipeline exceptions can be triggered. This requires extra complexity in the processor core to track the address of each instruction as it progresses through the pipeline.

This approach behaves the same if implemented as a single-scalar machine, or as a super-scalar machine.

On entry to an exception:

- The processor will be in Supervisor Mode.
- LR points to the next instruction to be executed when the Handler finishes.
- For the Trap exception, R1 will be set to the argument of the `trap` instruction.
- For all other exceptions, the address of the instruction that caused the exception is placed in R1.
- The contents of the FLAGS register just before the exception will be copied into SFLAGS_S.
- Interrupts are disabled.

For most exceptions on entry to the handler R1 will point to the instruction that caused the exception. This is necessary because instructions are variable length and it is a non-trivial task to determine the start of the instruction that caused the exception without

walking back through the instruction stream. In some circumstances this could cause other exceptions as well (e.g., if the instruction that caused the exception was the first instruction in a code segment then accessing the previous word could cause a memory protection exception).

Exceptions can only happen in User Mode and cause a switch to Supervisor Mode. In the Supervisor, NMI and Interrupt Modes exceptions have no effect (the `trap` instruction behaves as a `nop`).

The complete list of exceptions is given in section 6.7.

The core processes exceptions in the following manner:

```
if exception thrown && FLAGS.Mode == UserMode) then
      SFLAGS_S = FLAGS
      LR_S     = PC
      if (ExceptionNum is TRAP) then
            R1_S   = instruction argument
      else
            R1_S   = LastPC
      endif
      FLAGS.Mode  = SupervisorMode
      FLAGS.E     = 0
      PC          = (ExceptionNum * 4) + IVTB
endif
```

Returning from an exception is done by the `rtie` instruction (see section 6.4 below).

The null exception handler is simply the `rtie` instruction alone, which can be accommodated within the Interrupt Vector Table (see section 6.6 below).

## 6.3 Interrupt Processing

The processor has a single IRQ input and a single NMI input. If interrupts are enabled, activating the IRQ input will cause the processor to switch to Interrupt Mode when the current instruction finishes and query the interrupt controller for an interrupt number, IRQ_NUM. This number is converted in to a word offset, and is then used to index (plus offset 0x0040) into the IVT to point to the interrupt handler.

Non-Maskable Interrupts can occur at any time and cannot be disabled. The additional shadow registers in NMI mode ensure that an NMI handler can restore the processor to its state immediately prior to the NMI event.

The interrupt vector controller is responsible for implementing any priority based schemes.

The number of interrupt vectors is limited by the width of the irq_num bus. The irq_num bus is 8-bit and so supports up to 256 different interrupt vectors.

On entry to an interrupt:

- The processor will be in Interrupt (or NMI) Mode.
- LR points to the next instruction to be executed when the handler finishes.
- The contents of the FLAGS register just before the interrupt will be copied into SFLAGS_I (or SFLAGS_N for NMI mode).
- Interrupts are disabled.

The core processes interrupts in the following manner:

```
if (posedge NMI) then
      SFLAGS_N     = FLAGS
      FLAGS.Mode   = NMIMode
      LR_N         = PC
      PC           = 0x0004 + IVTB
elseif (FLAGS.E && IRQ) then
      SFLAGS_I     = FLAGS
      FLAGS.E      = 0
      FLAGS.Mode   = InterruptMode
      LR_I         = PC
      PC           = 0x40 + (IRQ_NUM * 4) + IVTB
endif
```

## 6.4 Returning from Interrupts and Exceptions

The `rtie` instruction is used to return from interrupts and exceptions and is implemented as:

```
if (FLAGS.Mode == NMIMode) then
      FLAGS = SFLAGS_N
      PC    = LR_N
elseif (FLAGS.Mode == InterruptMode) then
      FLAGS = SFLAGS_I
      PC    = LR_I
elseif (FLAGS.Mode == SupervisorMode) then
      FLAGS = SFLAGS_S
      PC    = LR_S
endif
```

Thus the null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code.

## 6.5 Nested Interrupts

To support nestable interrupts some additional work is required of the handler to save, and later restore, the processor state.

On entry:

```
push        {%r1-%r14}, #4     // Save registers and allocate
                               //  one word on the stack
movs2r      %r1, %flags_i      // Get SFLAGS_I
st          %r1, @(0,%r15)     // And save on the stack
irqe                           // Re-enable interrupts
```

Then on exit we restore the original registers and flags:

```
irqd                                    // Disable interrupts
ld          %r1, @(0,%r15)              // Get SFLAGS_I from stack
movr2s      %flags_i, %r1               // Restore SFLAGS_I
ldm         {%r1-%r5}, @(4, %r15)       // Restore R1-R5
pop         {%r6-%r14}, #24             // Restore registers
                                        //  & free stack space
rtie                                    // Return from interrupt
```

Note that the exact register lists for the push and pop can be tailored for specific handlers to minimise stack usage and interrupt response time.

NMI handlers that are expected to return control to the interrupted process may use similar entry and exit code, with the omission of the `irqe` and `irqd` instructions.

## 6.6 Interrupt Vector Table

The Interrupt Vector Table (IVT) contains instructions. These will often be bra.p to the relevant handler routines. The IVT manages both exceptions and interrupts. The IVT sits in memory with its base defined by IVTB.

IVTB resets to IVTB_reset_value[31:12]. This is a 20-bit hardware input to the XAP3 whose value is determined in the ASIC Verilog, This allows the IVTB to have reset values on 4kB steps. This means that reset will set the PC register to :

- PC[31:12] = IVTB_reset_value[31:12]
- PC[11:0] = 0

The IVTB register defines the base address of the IVT. This allows different tasks to have different handlers or for the IVT to be moved to RAM.

The start address of the IVT must be 32-bit aligned. Each entry in the table is one 32-bit word, and contains executable code. This is typically a branch to the start of the actual handler, but may simply be the null handler (the `rtie` instruction).

The layout of the IVT is shown below. The unused exceptions are for future expansion.

| *Offset* | *Processor Mode* | *Vector* | |
|---|---|---|---|
| 0x0000 | S | 0 | Reset |
| 0x0004 | N | 1 | Non-Maskable Interrupt |
| 0x0008 | S | 2 | Trap Exception |
| 0x000C | S | 3 | SingleStep Exception |
| 0x0010 | S | 4 | Break Exception |
| 0x0014 | S | 5 | UnknownInstruction Exception |
| 0x0018 | S | 6 | *reserved* Exception |
| 0x001C | S | 7 | *reserved* Exception |
| 0x0020 | S | 8 | DivideByZero Exception |
| 0x0024 | S | 9 | NullPointer Exception |
| 0x0028 | S | 10 | IllegalInstruction Exception |
| 0x002C | S | 11 | IllegalInstructionAccess Exception |
| 0x0030 | S | 12 | IllegalDataAccess Exception |
| 0x0034 | S | 13 | IllegalCodeAddress Exception |
| 0x0038 | S | 14 | *reserved* Exception |
| 0x003C | S | 15 | *reserved* Exception |
| 0x0040 | I | 0 | Int00 Interrupt |
| 0x0044 | I | 1 | Int01 Interrupt |
| 0x0048 | I | 2 | Int02 Interrupt |
| ... | ... | ... | ... |

## 6.7 Exceptions

Further details of each exception are described below. It is expected that the OS will kill (and maybe restart) applications throwing exceptions 8 (DivideByZero) onwards.

### *6.7.1 Reset*

While not technically an exception, placing the reset vector in the exception group leads to a simpler processor model.

### *6.7.2 NMI*

The NMI handler is called when an edge is detected on the NMI input. This would be used for system-critical events that cannot be ignored (unlike ordinary interrupts that can be disabled) such as loss of power, fatal hardware failure, etc.

### *6.7.3 Trap*

Software trap instructions may be used to call an operating system. All `trap` instructions take an argument, either a register or an immediate. On entry to the trap handler, LR points to the next instruction, and R1 is set to the argument of the instruction (e.g., OS function number).

The following instructions cause this exception

- `trap.r`
- `trap.i`

### 6.7.4 SingleStep

When the T bit is set in the FLAGS register, this exception is triggered after every User Mode instruction, and only if that instruction has not caused any other exception (e.g., DivideByZero). The LR points to the next instruction to be executed and R1 points to the instruction that caused the exception. This supports remote debugging and single stepping of User Mode applications.

### 6.7.5 Break

The Break exception can be caused by either a brk instruction or by a break condition being satisfied in the BRK registers.

Break events are handled according to this logic.

```
if FLAGS.Mode == User && FLAGS.B == 1 then
      throw break exception
else if debug enabled then
      halt
else
      nop
endif
```

Debug mode is enabled using the SIF. The B flag takes precedence over the debug mode action.

On the simulator, debug mode is always enabled and any break conditions will halt the processor unless the processor is in User Mode and the B flag is set. Similarly, the emulator will always make sure that the processor is in debug mode, by using the appropriate SIF command.

The `brk` instructions are not protected instructions. An operating system can prevent a User Mode task from stopping the processor by setting the B bit of the FLAGS register. For normal debugging of User Mode code the T and B flags are not set and the Simulator and Emulator will use instruction replacement and the BRK registers to implement breakpoints.

### 6.7.6 UnknownInstruction

All opcodes that do not correspond to a valid instruction trigger this exception.

LR points to the next instruction to be executed and R1 points to the undefined instruction just executed.

This exception has two useful functions. Firstly, it can quickly trap bad code if it causes the processor to start executing data or other non-code.

Secondly, and perhaps more importantly, it is a simple, proven, mechanism for supporting pseudo-instructions in a processor. This has two significant benefits:

- It allows any version of the XAP3 core to support, through software emulation, any custom instruction of any other version of the XAP3 core, albeit at a slower rate of execution. Note this includes dynamically updating a system with new instructions simply by adding more functionality to the instruction emulation code.
- It supports experimentation and analysis of any new instruction simply by adding emulation code, perhaps to evaluate code size or feature exploration.

### 6.7.7 DivideByZero

All forms of divide and remainder instructions check the value of the denominator. If it is zero, then this triggers a DivideByZero exception. LR points to the next instruction to be executed and R1 points to the divide instruction just executed.

The following instructions can cause this exception:

- `div.u.r`
- `div.s.r`
- `rem.u.r`
- `rem.s.r`

### 6.7.8 NullPointer

The xap3_mmu specifies a range of addresses starting from 0 to be treated as a NULL pointer. The rationale behind this is that pointer objects are often pointers to structures and therefore it is not sufficient to simply detect accesses to address zero as a NULL pointer condition.

LR points to the next instruction to be executed and R1 points to the instruction just executed.

The following instructions can cause this exception:

- `ld.i, ld.16s.i, ld.16z.i, ld.8.s.i, ld.8.z.i`
- `ld.r, ld.16s.r, ld.16z.r, ld.8.s.r, ld.8.z.r`
- `ldm, ldm.x`
- `st.i, st.16.i, st.8.i`

- `st.r, st.16.r, st.8.r`
- `stm, stm.x`
- `push, push.r`
- `pop, pop.ret`

Note: `ldm.u` and `stm.u` do not generate an exception, because they are not valid instructions in user mode.

### 6.7.9 IllegalInstruction

The XAP3 core will execute most code in User Mode. This mode is restricted in its access rights to certain instructions; executing any protected instruction in User Mode will trigger this exception. LR points to the next instruction to be executed and R1 points to the instruction that caused the exception.

The following instructions cause this exception, because they provide access to the FLAGS register and could be used to create mode changes:

- `rtie`
- `ldm.u`
- `stm.u`
- `movr2s, movs2r`
- `movr2b, movb2r`
- `irqe, irqd`

The following instructions cause this exception, because they alter the behaviour CPU core.

- `sleepsif, sleepnop`
- `halt`
- `sif`

The `sif` instruction is not allowed in User Mode, because it can alter the timing of accesses from external devices to the registers and memory of the processor. Systems wishing to allow User Mode to control when SIF accesses can take place, would need to provide a Supervisor Mode routine accessible via the trap instruction.

### 6.7.10 IllegalInstructionAccess and IllegalDataAccess

These exceptions are triggered by an external Memory Management Unit (xap3_mmu) in response to the XAP3 core accessing memory that violates the current access rights (e.g., reading from memory that does not belong to the currently executing process, or writing to memory that is tagged as read-only).

IllegalInstructionAccess is thrown when an instruction fetch violates the access rights. IllegalDataAccess is thrown when an instructions data read or write violates the access rights.

On entry to the handler LR points to the next instruction to be executed and R1 points to the instruction that caused the exception. It is intended that the xap3_mmu places the address that caused the exception in some memory mapped register accessible to the exception handler code.

These exceptions are important in virtual memory systems (implemented in software with an appropriate hardware MMU). When a program accesses data (or code) in its memory space that currently belongs to another task, this exception will be triggered since it violates the memory protection afforded to the other task. The virtual memory manager can then swap out the sleeping task's memory chunk, swap in the active task's memory chunk, and then restart the current instruction.

IllegalDataAccess can be caused by the following instructions:

- `ld.i, ld.16s.i, ld.16z.i, ld.8.s.i, ld.8.z.i`
- `ld.r, ld.16s.r, ld.16z.r, ld.8.s.r, ld.8.z.r`
- `ldm, ldm.x`
- `st.i, st.16.i, st.8.i`
- `st.r, st.16.r, st.8.r`
- `stm, stm.x`
- `push, push.r`
- `pop, pop.ret`

Note: `ldm.u` and `stm.u` do not generate an exception, because they are not valid instructions in user mode.

### *6.7.11 IllegalCodeAddress*

All instructions are half-word aligned, such that PC[0] = 0. This exception is triggered when an odd value is written to PC, (i.e., PC[0] = 1). LR points to the next instruction to be executed (i.e., the invalid address) and R1 points to the branch instruction that caused the exception.

The following instructions can cause this exception:

- `bra.r`
- `bsr.r`
- `pop.ret`
- `rtie`

Note: it is not possible for the immediate forms of the branch instructions throw this exception, because they cannot encode illegal addresses.

# 7 Addressing Modes

ALU instructions have 2 addressing modes:

- Register, Register, Immediate
- Register, Register, Register

In each case the result is put in a register. The registers are always 32-bit.

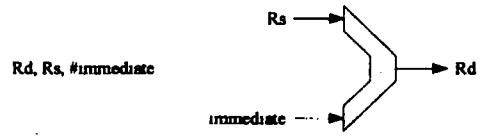
The only instructions that can read memory are ld.*, ldm.*, pop.*, swap.r.

The only instructions that can write to memory are st.*, stm.*, push.*, swap.r.

Hence the XAP3 has a load-store architecture.

## 7.1 Register, Register, Immediate

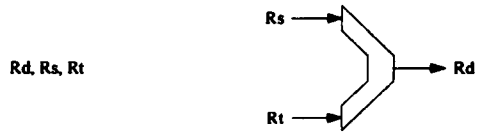
One input to the ALU comes from a register and one from an immediate value that is part of the instruction word. The input and output register can be specified separately.




The immediate is normally 17-bit and can be aligned low (.i) or high (.h) in the 32-bit word.

## 7.2 Register, Register, Register

Both inputs and the output can be specified separately.



## 7.3 Load, Store

There are two forms of the basic load and store instructions.

### *7.3.1 Displacement Addressing*

Displacement Addressing is implemented by the '.i' form of the load and store instructions. The addresses is calculated as:

offset + Ra

### *7.3.2 Indexed Addressing*

Indexed Addressing is implemented by the '.r' form of the load and store instructions. The addresses is calculated as:

(Rx * n) + Ra

| | |
|---|---|
| 8-bit data | n = 1 |
| 16-bit data | n = 2 |
| 32-bit data | n = 4 |

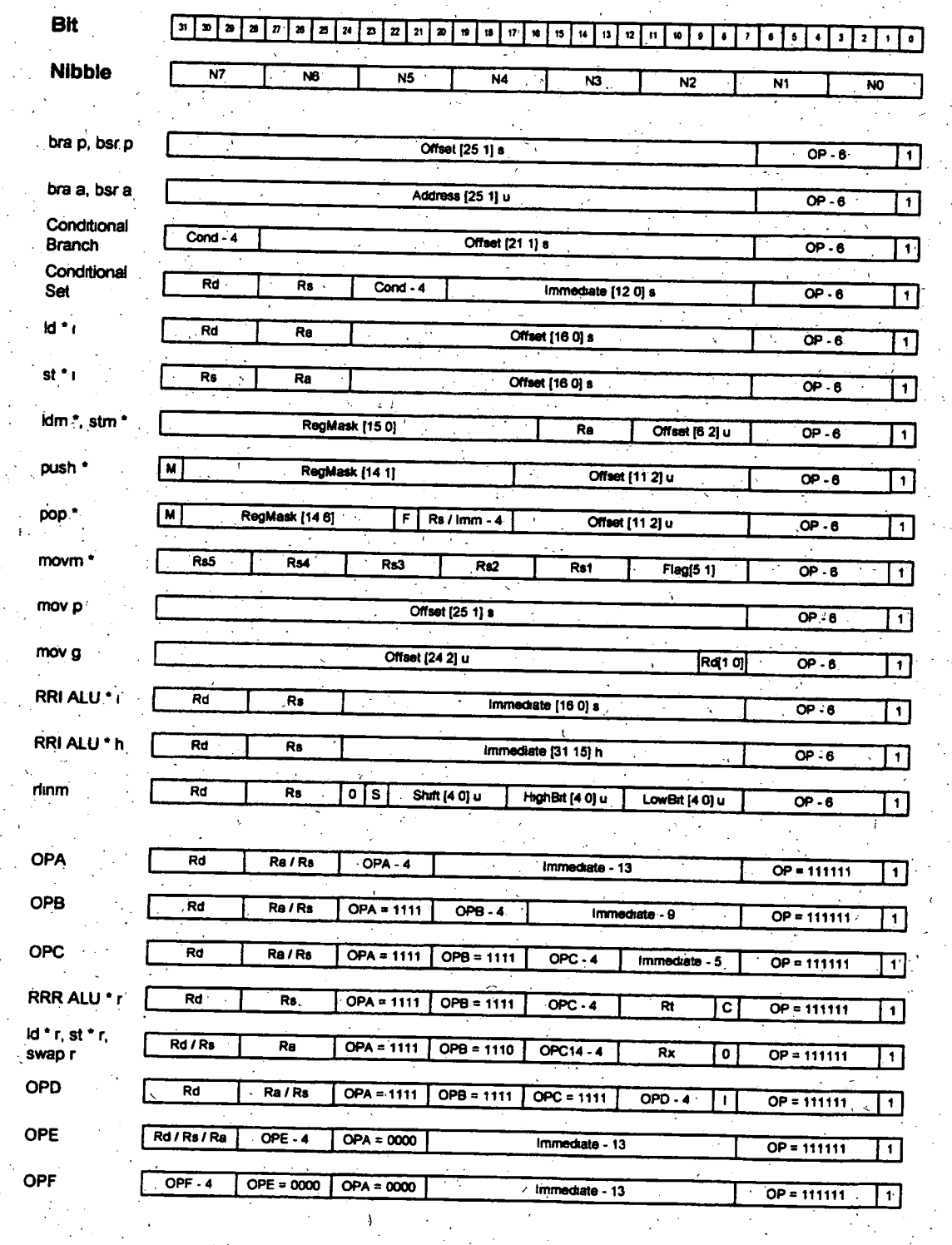
# 8 Instruction Set

## 8.1 Notation Conventions

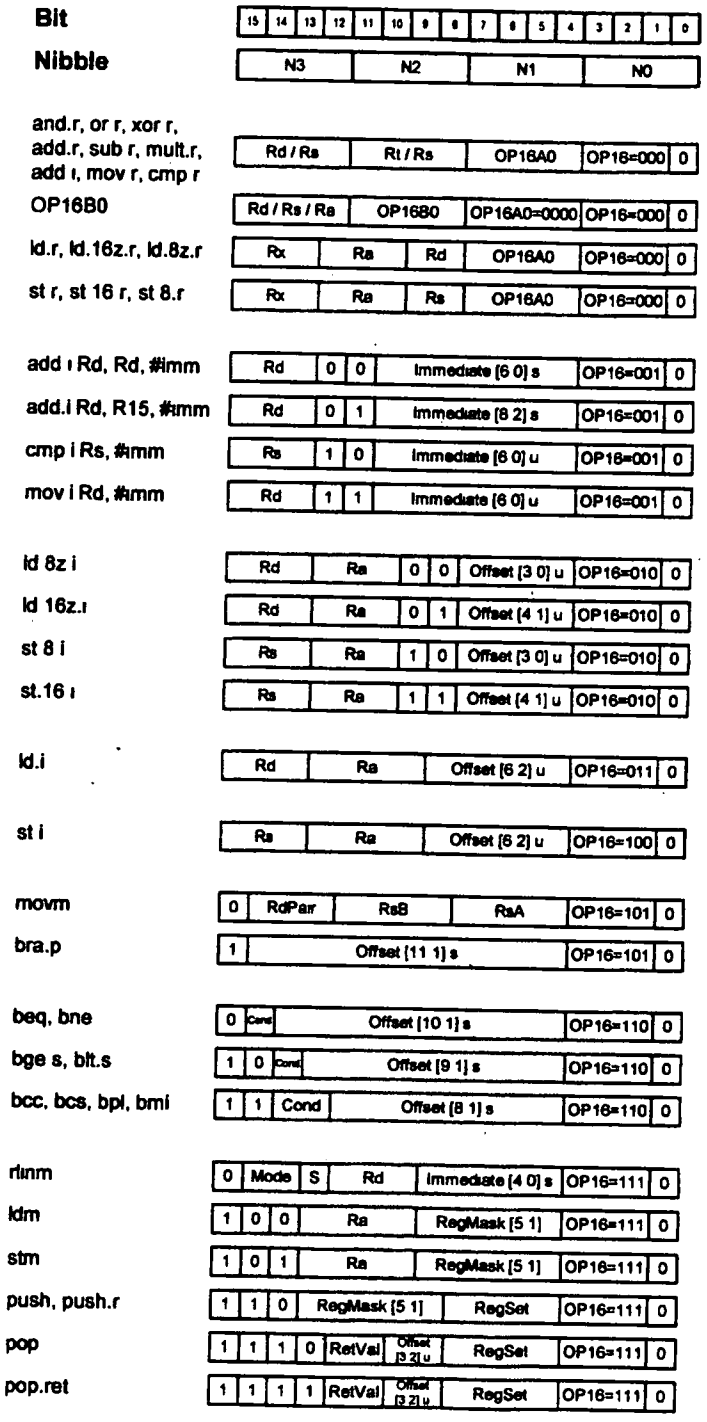
In the following sections the following conventions are used to describe various aspects of the instruction set.

- Instruction parameters are prefixed by $. More details on Parameters in section 2.5.2.6.
- Registers are prefixed by %.
- Rd is the destination register of the instruction
- Rs, and Rt are the primary and secondary registers containing source operands
- Ra, and Rx are registers containing a base and index address for an operand. An offset may also be added to the register contents to form the actual address of the operand. Ra is a byte address for the base. Rx is a scaled address for the index (*4 for 32-bit, *2 for 16-bit, *1 for 8-bit data). Both Ra and Rx are unsigned numbers.
- Immediate[] indicates a constant operand encoded as part of the instruction. In the documentation #immediate[highbit:lowbit]s or #immediate[highbit:lowbit]u is used to distinguish between signed and unsigned immediates. Immediates are prefixed by #. Note, the # is not used for address offsets (e.g @(0,%r1))
- Address[] indicates an absolute address. This is a byte-address, unless specified otherwise. Addresses are prefixed by @. Absolute addresses are always interpreted as 32-bit unsigned numbers.
- Base addresses stored in Ra should be interpreted as byte addresses. Index offsets stored in Rx should be multiplied by (1, 2, 4) respectively for (8, 16, 32) bit data to get the desired byte address offset.
- Offset[] indicates an address offset. This may be relative to the program counter (in branch instructions, for example), or to a pointer read from a register (the stack pointer, for example). Offsets can be signed or signed numbers (indicated by appended s or u). Offsets are byte-address values, unless specified otherwise. In the documentation offset[highbit:lowbit]s or offset[highbit:lowbit]u is used to distinguish between signed, unsigned and index offsets. A signed offset should be sign extended to a 32-bit word before calculating an address. An unsigned offset should be zero extended to a 32-bit word before calculating an address.

## 8.2 Structure of 32-bit Instructions



## 8.3 Structure of 16-bit Instructions



## 8.4 Base Instructions

The table below is an alphabetical list of the mnemonics representing all the instructions in XAP3. For each mnemonic there are details of the operands specified in the assembly language and a brief description of the instruction.

All instructions have a 32-bit form. The 16-bit instructions are a sub-set of the 32-bit instructions. Most of the 16-bit instructions implement a sub-set of the functionality of their 32-bit forms.

The 3 privileged modes (NMI, Interrupt, Supervisor) can use all of these instructions. A subset of these instructions are available in User mode. Section 6.7.9 lists the instructions that are not available in User mode.

| *Mnemonic* | *Operands* | *Operation* | *Min Op* |
|---|---|---|---|
| abs.r | Rd, Rs | Absolute Register | 32 |
| acc.64.r | Rd, Rs | Accumulate (64-bit) | 32 |
| add.c.r | Rd, Rs, Rt | Add Register with Carry | 32 |
| add.h | Rd, Rs, #immediate[31:15]h | Add High Immediate | 32 |
| add.i | Rd, Rs, #immediate[16:0]s | Add Immediate | 16 |
| add.r | Rd, Rs, Rt | Add Register | 16 |
| and.h | Rd, Rs, #immediate[31:15]h | AND High Immediate | 32 |
| and.i | Rd, Rs, #immediate[16:0]s | AND Immediate | 32 |
| and.r | Rd, Rs, Rt | AND Register | 16 |
| bcase.2 | Rs, #immediate[12:0]u | Branch Case (2 byte table) | 32 |
| bcase.4 | Rs, #immediate[12:0]u | Branch Case (4 byte table) | 32 |
| bcc | #offset[21:1]s | Branch if Carry Clear | 16 |
| bcs | #offset[21:1]s | Branch if Carry Set | 16 |
| beq | #offset[21:1]s | Branch if Equal | 16 |
| bge.s | #offset[21:1]s | Branch if Greater Than or Equal (Signed) | 16 |
| bgt.s | #offset[21:1]s | Branch if Greater Than (Signed) | 32 |
| bgt.u | #offset[21:1]s | Branch if Greater Than (Unsigned) | 32 |
| ble.s | #offset[21:1]s | Branch if Less Than or Equal (Signed) | 32 |
| ble.u | #offset[21:1]s | Branch if Less Than (Unsiged) | 32 |
| blt.s | #offset[21:1]s | Branch if Less Than (Signed) | 16 |
| bmi | #offset[21:1]s | Branch if Minus | 32 |
| bne | #offset[21:1]s | Branch if Not Equal | 16 |
| bpl | #offset[21:1]s | Branch if Plus | 32 |
| bra.a | #address[25:1]u | Branch Absolute | 32 |
| bra.p | #offset[25:1]s | Branch PC Relative | 16 |
| bra.r | Rs | Branch Register | 16 |
| brk | | Break | 16 |

| *Mnemonic* | *Operands* | *Operation* | *Min Op* |
|---|---|---|---|
| bsr.a | #address[25:1]u | Branch Subroutine Absolute | 32 |
| bsr.p | #offset[25:1]s | Branch Subroutine Relative | 32 |
| bsr.r | Rs | Branch to Subroutine Register | 16 |
| bvc | #offset[21:1]s | Branch if Overflow Clear | 32 |
| bvs | #offset[21:1]s | Branch if Overflow Set | 32 |
| cmp.16.h | Rs, #immediate[31:16] | Compare High Immediate | 32 |
| cmp.16.i | Rs, #immediate[15:0] | Compare Immediate | 32 |
| cmp.16.r | Rs, Rt | Compare Register | 32 |
| cmp.16c.h | Rs, #immediate[31:16] | Compare with Carry High Immediate | 32 |
| cmp.16c.i | Rs, #immediate[15:0] | Compare with Carry Immediate | 32 |
| cmp.16c.r | Rs, Rt | Compare with Carry Register | 32 |
| cmp.16x.h | Rs, #immediate[31:16] | Compare High Immediate | 32 |
| cmp.16x.i | Rs, #immediate[15:0] | Compare Immediate | 32 |
| cmp.16xc.h | Rs, #immediate[31:16] | Compare with Carry High Immediate | 32 |
| cmp.16xc.i | Rs, #immediate[15:0] | Compare with Carry Immediate | 32 |
| cmp.8.i | Rs, #immediate[7:0] | Compare Immediate | 32 |
| cmp.8.r | Rs, Rt | Compare Register | 32 |
| cmp.8c.i | Rs, #immediate[7:0] | Compare with Carry Immediate | 32 |
| cmp.8c.r | Rs, Rt | Compare with Carry Register | 32 |
| cmp.8x.i | Rs, #immediate[7:0] | Compare Immediate | 32 |
| cmp.8xc.i | Rs, #immediate[7:0] | Compare with Carry Immediate | 32 |
| cmp.c.h | Rs, #immediate[31:15]h | Compare with Carry High Immediate | 32 |
| cmp.c.i | Rs, #immediate[16:0]s | Compare with Carry Immediate | 32 |
| cmp.c.r | Rs, Rt | Compare with Carry Register | 32 |
| cmp.h | Rs, #immediate[31:15]h | Compare High Immediate | 32 |
| cmp.i | Rs, #immediate[16:0]s | Compare Immediate | 16 |
| cmp.r | Rs, Rt | Compare Register | 16 |
| cmp.x.h | Rs, #immediate[31:15]h | Compare High Immediate | 32 |
| cmp.x.i | Rs, #immediate[16:0]s | Compare Immediate | 32 |
| cmp.xc.h | Rs, #immediate[31:15]h | Compare with Carry High Immediate | 32 |
| cmp.xc.i | Rs, #immediate[16:0]s | Compare with Carry Immediate | 32 |
| div.s.r | Rd, Rs, Rt | Divide Register (Signed) | 16 |
| div.u.r | Rd, Rs, Rt | Divide Register (Unsigned) | 16 |
| flip.16b.r | Rd, Rs, Rt, #lowbyte2 | Flip and Byte Swap (in 16-bit words) | 32 |
| flip.16x.r | Rd, Rs, Rt, #lowbyte2 | Flip and Bit Swap (in 16-bit words) | 32 |
| flip.8x.r | Rd, Rs, Rt, #lowbyte2 | Flip and Bit Swap (in bytes) | 32 |

| *Mnemonic* | *Operands* | *Operation* | *Min Op* |
|---|---|---|---|
| flip.b.r | Rd, Rs, Rt, #lowbyte2 | Flip and Byte Swap | 32 |
| flip.r | Rd, Rs, Rt, #lowbyte2 | Flip | 32 |
| flip.x.r | Rd, Rs, Rt, #lowbyte2 | Flip and Bit Swap | 32 |
| halt | | Halt | 32 |
| irqd | | Disable Interrupts | 32 |
| irqe | | Enable Interrupts | 32 |
| irqs | Rd | Interrupt Status | 32 |
| ld.16s.i | Rd, @(offset[16:0]s, Ra) | Load with Displacement Addressing (16-bit Sign Extend) | 32 |
| ld.16s.r | Rd, @(Rx, Ra) | Load with Indexed Addressing (16-bit Sign Extend) | 32 |
| ld.16z.i | Rd, @(offset[16:0]s, Ra) | Load with Displacement Addressing (16-bit, Zero Extend) | 16 |
| ld.16z.r | Rd, @(Rx, Ra) | Load with Indexed Addressing (16-bit, Zero Extend) | 32 |
| ld.8s.i | Rd, @(offset[16:0]s, Ra) | Load with Displacement Addressing (8-bit Sign Extend) | 32 |
| ld.8s.r | Rd, @(Rx, Ra) | Load with Indexed Addressing (8-bit Sign Extend) | 32 |
| ld.8z.i | Rd, @(offset[16:0]s, Ra) | Load with Displacement Addressing (8-bit, Zero Extend) | 16 |
| ld.8z.r | Rd, @(Rx, Ra) | Load with Indexed Addressing (8-bit, Zero Extend) | 32 |
| ld.i | Rd, @(offset[16:0]s, Ra) | Load with Displacement Addressing | 16 |
| ld.r | Rd, @(Rx, Ra) | Load with Indexed Addressing | 32 |
| ldm | {RegList[15:0]}, @(offset[6:2]u, Ra) | Load Multiple | 16 |
| ldm.u | {RegList[15:0]}, @(offset[6:2]u, Ra) | Load Multiple (User) | 32 |
| ldm.x | {RegList[15:0]}, @(offset[6:2]u, Ra) | Load Multiple (Exchanged Order) | 32 |
| mov.g | Rd, #offset[24:2]u | Move GP Relative Address to Register | 32 |
| mov.p | Rd, #offset[25:1]s | Move PC Relative Address to Register | 32 |
| movb2r | Rd, Bs | Move Breakpoint to Register | 32 |
| movm | #imm[3:0]u/Rs1, #imm[3:0]u/Rs2, #imm[3:0]u/Rs3, #imm[3:0]u/Rs4, #imm[3:0]u/Rs5 | Move Multiple | 16 |

| *Mnemonic* | *Operands* | *Operation* | *Min Op* |
|---|---|---|---|
| movm.x | #imm[3:0]u/Rs1, #imm[3:0]u/Rs2, #imm[3:0]u/Rs3, #imm[3:0]u/Rs4, #imm[3:0]u/Rs5 | Move Multiple (Exchanged Order) | 32 |
| movr2b | Bd, Rs | Move Register to Breakpoint | 32 |
| movr2s | Sd, Rs | Move Register to Special | 32 |
| movs2r | Rd, Ss | Move Special to Register | 32 |
| msbit.r | Rd, Rs | Most Significant Bit Register | 32 |
| mult.16s.i | Rd, Rs, #immediate[15:0] | Multiply Immediate (16-bit Signed) | 32 |
| mult.16s.r | Rd, Rs, Rt | Multiply Register (16-bit Signed) | 32 |
| mult.16u.i | Rd, Rs, #immediate[15:0] | Multiply Immediate (16-bit Unsigned) | 32 |
| mult.16u.r | Rd, Rs, Rt | Multiply Register (16-bit Unsigned) | 32 |
| mult.sh.r | Rd, Rs, Rt, #immediate[5:0]u | Signed Multiply Register, then signed shift right by Immediate | 32 |
| mult.i | Rd, Rs, #immediate[16:0]s | Multiply Immediate | 32 |
| mult.r | Rd, Rs, Rt | Multiply Register | 16 |
| nop | | No Operation | 16 |
| or.h | Rd, Rs, #immediate[31:15]h | OR High Immediate | 32 |
| or.i | Rd, Rs, #immediate[16:0]s | OR Immediate | 16 |
| or.r | Rd, Rs, Rt | OR Register | 16 |
| pop | {RegList[14:6]}, #offset[11:2]u | Pop from Stack | 16 |
| pop.ret | #imm[3:0]u/Rs, {RegList[14:6]}, #offset[11:2]u | Pop from Stack and Return | 16 |
| print.r | Rs | Print Register | 32 |
| push | {RegList[14:1]}, #offset[11:2]u | Push to Stack | 16 |
| push.r | {RegList[14:1]}, #offset[11:2]u | Push to Stack and Registers | 16 |
| rem.s.r | Rd, Rs, Rt | Remainder Register (Signed) | 16 |
| rem.u.r | Rd, Rs, Rt | Remainder Register (Unsigned) | 32 |
| rlinm.s | Rd, Rs, #shift5u, #highbit5u, #lowbit5u | Rotate Left and Mask (Sign Extend) | 16 |
| rlinm.z | Rd, Rs, #shift5u, #highbit5u, #lowbit5u | Rotate Left and Mask (Zero Extend) | 16 |
| rotatel.r | Rd, Rs, Rt | Rotate Left Register | 32 |
| rtie | | Return from Interrupt/Exception | 32 |
| scc.i | Rd, Rs, #immediate[12:0]s | Set if Carry Clear | 32 |

| *Mnemonic* | *Operands* | *Operation* | *Min Op* |
|---|---|---|---|
| scc.r | Rd, Rs, Rt | Set if Carry Clear | 32 |
| scs.i | Rd, Rs, #immediate[12:0]s | Set if Carry Set | 32 |
| scs.r | Rd, Rs, Rt | Set if Carry Set | 32 |
| seq.i | Rd, Rs, #immediate[12:0]s | Set if Equal | 32 |
| seq.r | Rd, Rs, Rt | Set if Equal | 32 |
| sge.s.i | Rd, Rs, #immediate[12:0]s | Set if Greater Than or Equal (Signed) | 32 |
| sge.s.r | Rd, Rs, Rt | Set if Greater Than or Equal (Signed) | 32 |
| sgt.s.i | Rd, Rs, #immediate[12:0]s | Set if Greater Than (Signed) | 32 |
| sgt.s.r | Rd, Rs, Rt | Set if Greater Than (Signed) | 32 |
| sgt.u.i | Rd, Rs, #immediate[12:0]s | Set if Greater Than (Unsigned) | 32 |
| sgt.u.r | Rd, Rs, Rt | Set if Greater Than (Unsigned) | 32 |
| shiftl.64.i | Rd, #immediate[5:0]u | Shift Left Register (64-bit) | 32 |
| shiftl.64.r | Rd, Rt | Shift Left Register (64-bit) | 32 |
| shiftl.r | Rd, Rs, Rt | Shift Left Register | 16 |
| shiftr.64s.i | Rd, #immediate[5:0]u | Shift Right Immediate (64-bit Signed) | 32 |
| shiftr.64s.r | Rd, Rt | Shift Right Register (64-bit Signed) | 32 |
| shiftr.s.r | Rd, Rs, Rt | Shift Right Register (Signed) | 16 |
| shiftr.u.r | Rd, Rs, Rt | Shift Right Register (Unsigned) | 16 |
| sif | | SIF | 16 |
| sle.s.i | Rd, Rs, #immediate[12:0]s | Set if Less Than or Equal (Signed) | 32 |
| sle.s.r | Rd, Rs, Rt | Set if Less Than or Equal (Signed) | 32 |
| sle.u.i | Rd, Rs, #immediate[12:0]s | Set if Less Than (Unsiged) | 32 |
| sle.u.r | Rd, Rs, Rt | Set if Less Than (Unsiged) | 32 |
| sleepnop | | Sleep | 32 |
| sleepsif | | Sleep and Allow SIF | 32 |
| slt.s.i | Rd, Rs, #immediate[12:0]s | Set if Less Than (Signed) | 32 |
| slt.s.r | Rd, Rs, Rt | Set if Less Than (Signed) | 32 |
| smi.i | Rd, Rs, #immediate[12:0]s | Set if Minus | 32 |
| smi.r | Rd, Rs, Rt | Set if Minus | 32 |
| sne.i | Rd, Rs, #immediate[12:0]s | Set if Not Equal | 32 |
| sne.r | Rd, Rs, Rt | Set if Not Equal | 32 |
| spl.i | Rd, Rs, #immediate[12:0]s | Set if Plus | 32 |
| spl.r | Rd, Rs, Rt | Set if Plus | 32 |
| st.16.i | Rs, @(offset[16:0]s, Ra) | Store with Displacement Addressing (16-bit) | 16 |
| st.16.r | Rs, @(Rx, Ra) | Store with Indexed Addressing (16-bit) | 32 |
| st.8.i | Rs, @(offset[16:0]s, Ra) | Store with Displacement | 16 |

| *Mnemonic* | *Operands* | *Operation* | *Min Op* |
|---|---|---|---|
| | | Addressing (8-bit) | |
| st.8.r | Rs, @(Rx, Ra) | Store with Indexed Addressing (8-bit) | 32 |
| st.i | Rs, @(offset[16:0]s, Ra) | Store with Displacement Addressing | 16 |
| st.r | Rs, @(Rx, Ra) | Store with Indexed Addressing | 32 |
| stm | {RegList[15:0]}, @(offset[6:2]u, Ra) | Store Multiple | 16 |
| stm.u | {RegList[15:0]}, @(offset[6:2]u, Ra) | Store Multiple (User) | 32 |
| stm.x | {RegList[15:0]}, @(offset[6:2]u, Ra) | Store Multiple (Exchanged Order) | 32 |
| sub.c.r | Rd, Rs, Rt | Subtract Register with Carry | 32 |
| sub.r | Rd, Rs, Rt | Subtract Register | 16 |
| sub.x.h | Rd, Rs, #immediate[31:15]h | Subtract High Immediate (Reverse Order) | 32 |
| sub.x.i | Rd, Rs, #immediate[16:0]s | Subtract Immediate (Reverse Order) | 32 |
| svc.i | Rd, Rs, #immediate[12:0]s | Set if Overflow Clear | 32 |
| svc.r | Rd, Rs, Rt | Set if Overflow Clear | 32 |
| svs.i | Rd, Rs, #immediate[12:0]s | Set if Overflow Set | 32 |
| svs.r | Rd, Rs, Rt | Set if Overflow Set | 32 |
| swap.r | Rd, @(Rx, Ra) | Swap Register with Memory | 32 |
| trap.i | #immediate[12:0]s | Trap Immediate | 32 |
| trap.r | Rs | Trap Register | 32 |
| xor.h | Rd, Rs, #immediate[31:15]h | XOR High Immediate | 32 |
| xor.i | Rd, Rs, #immediate[16:0]s | XOR Immediate | 16 |
| xor.r | Rd, Rs, Rt | XOR Register | 16 |

## 8.5 Aliased Instructions

Several instructions have aliases (alternate names) that may be written in an assembler source file and will be translated into the corresponding real instruction by the assembler.

| *Alias Mnemonic* | | *Real Instruction* | | *Min Op* |
|---|---|---|---|---|
| mov.h | Rd, #imm | or.h | Rd, %r0, #imm | 32 |
| mov.i | Rd, #imm | or.i | Rd, %r0, #imm | 16 |
| mov.r | Rd, Rt | or.r | Rd, %r0, Rt | 16 |
| blt.u | label | bcs | label | 16 |
| bge.u | label | bcc | label | 16 |
| slt.u.i | Rd, Rs, #imm | scs.i | Rd, Rs, #imm | 32 |

| | | | | |
|---|---|---|---|---|
| sge.u.i | Rd, Rs, #imm | scc.i | Rd, Rs, #imm | 32 |
| slt.u.r | Rd, Rs, Rt | scs.r | Rd, Rs, Rt | 32 |
| sge.u.r | Rd, Rs, Rt | scc.r | Rd, Rs, Rt | 32 |

## 8.6 Flag Usage Conventions

The four main arithmetic flags (Z, N, C, V) are set by various instructions and also as defined by the function calling convention.

The logical instructions set Z and N on the value written into the target register.

All load and move instructions set the Z and N flags. The multiple variants (ldm and movm) set the flags on the last load or move respectively.

All four flags (Z, N, C, V) are set by the result of arithmetic instructions (add, subtract, etc).

Shift and rotate instructions set the Z and N flags, and optionally the C flag depending on the specific instruction

Function calls set the Z and N flags depending on the value in R1, and set C and V to 0.

## 8.7 Interpretation of Immediates

### *8.7.1 Low Immediates*

Low Immediate instructions (Type = .i) have a 17-bit immediate, #immediate[16:0]. This should be interpreted as a signed number as follows :

- data[31:16] = #immediate[16] // repeated 16 times
- data[15:0] = #immediate[15:0]

The low immediate range is :

- -65536 // 0xFFFF0000
- 65535 // 0x0000FFFF

It can represent numbers such as :

- 0x00001234
- 0xFFFF4321

### *8.7.2 High Immediates*

High Immediate instructions (Type = .h) have a 17-bit immediate, #immediate[31:15]. This should be interpreted as follows :

- data[31:16] = #immediate[31:16]
- data[15:0] = #immediate[15] // repeated 16 times

This can represent numbers such as :

- 0x12340000
- 0x4321FFFF

### 8.7.3 Addresses

All instructions are 16 or 32-bit. They are arranged to start on 2byte boundaries (i.e at even addresses).

There are several instructions that pass 25-bit addresses or offsets, on 2byte boundaries :

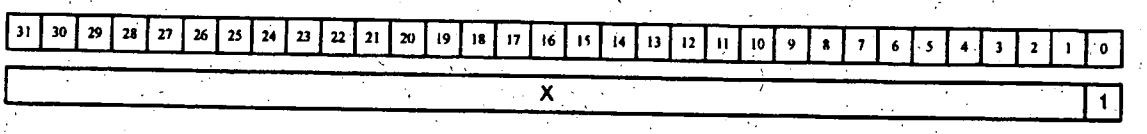
- bra.a, bsr.a // Both use unsigned absolute address[25:1]. address[0] = 0.
- bra.p, bsr.p // Both use signed PC-relative offset[25:1]. offset[0] = 0.
- mov.p // adds (PC + offset[25:1]s) to Rd
  // offset is interpreted as bits [25:1] (sign-extended). offset[0] = 0.
- mov.g // adds (GP + offset[24:2]u) to Rd
  // offset is interpreted as bits [24:2] (zero-extended). offset[1:0] = 00.

### 8.7.4 Register or Small Immediate Values

Some of the instructions allow a user to specify either a small immediate or a register (e.g movm). The instruction encoding of this operand uses five bits: one bit selects between register and immediate, four bits specify the register or the immediate.

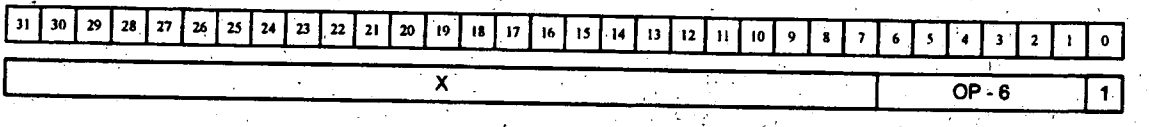
Since the XAP3 has a constant zero register we don't need to able to express an immediate value of zero, therefore the immediate value of zero is interpreted as –1. This allows users to specify an immediate in the range 1 to 15 and –1. If an immediate of zero is required the %r0 register must be specified.

## 8.8 32-bit Instruction Opcodes




Note: the 32-bit instruction 0xFFFFFFFF is a SIF instruction.

### *8.8.1 Opcode – OP[5:0]*




| *OP* | *OP binary* | *6-bit Mnemonic* | *25-bit Argument* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 00 0000 | push<br>push.r | $mode1, #mask14 ,<br>#offset[11:2]u (SP relative) | push: none<br>push.r: Z, N | n<br>n | 16<br>16 |
| 1 | 00 0001 | mov.p %r1 | offset[25:1]s (PC relative) | | 1 | 32 |
| 2 | 00 0010 | mov.p %r2 | offset[25:1]s (PC relative) | | 1 | 32 |
| 3 | 00 0011 | mov.p %r3 | offset[25:1]s (PC relative) | | 1 | 32 |
| 4 | 00 0100 | mov.p %r4 | offset[25:1]s (PC relative) | | 1 | 32 |
| 5 | 00 0101 | mov.p %r5 | offset[25:1]s (PC relative) | | 1 | 32 |
| 6 | 00 0110 | mov.p %r6 | offset[25:1]s (PC relative) | | 1 | 32 |
| 7 | 00 0111 | mov.p %r7 | offset[25:1]s (PC relative) | | 1 | 32 |
| 8 | 00 1000 | mov.p %r8 | offset[25:1]s (PC relative) | | 1 | 32 |
| 9 | 00 1001 | mov.p %r9 | offset[25:1]s (PC relative) | | 1 | 32 |
| 10 | 00 1010 | mov.p %r10 | offset[25:1]s (PC relative) | | 1 | 32 |
| 11 | 00 1011 | mov.p %r11 | offset[25:1]s (PC relative) | | 1 | 32 |
| 12 | 00 1100 | mov.p %r12 | offset[25:1]s (PC relative) | | 1 | 32 |
| 13 | 00 1101 | mov.p %r13 | offset[25:1]s (PC relative) | | 1 | 32 |
| 14 | 00 1110 | mov.p %r14 | offset[25:1]s (PC relative) | | 1 | 32 |
| 15 | 00 1111 | pop<br>pop.ret | $mode2, #mask9,<br>#imm[3:0]u/Rs, #offset[11:2]u<br>(SP relative) | Z, N, C, V<br>Set on %r1<br>value. i.e the<br>return value. | n<br>n | 16<br>16 |
| 16 | 01 0000 | mov.g<br>%r0-%r3 | offset[24:2]u (GP relative) | | 1 | 32 |
| 17 | 01 0001 | mov.g<br>%r4-%r7 | offset[24:2]u (GP relative) | | 1 | 32 |

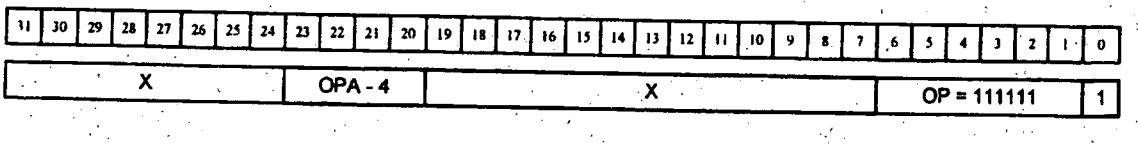
| *OP* | *OP binary* | *6-bit Mnemonic* | *25-bit Argument* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 18 | 01 0010 | mov.g %r8-%r11 | offset[24:2]u (GP relative) | | 1 | 32 |
| 19 | 01 0011 | mov.g %r12-%r15 | offset[24:2]u (GP relative) | | 1 | 32 |
| 20 | 01 0100 | movm | $flag5, Rs1, Rs2, Rs3, Rs4, Rs5<br>Copy order = R1 to R5 | Z, N<br>On highest reg changed | n | 16 |
| 21 | 01 0101 | movm.x | $flag5, Rs1, Rs2, Rs3, Rs4, Rs5<br>Copy order = R5 to R1 | Z, N<br>On lowest reg changed | n | 32 |
| 22 | 01 0110 | beq<br>bne<br>bmi<br>bpl<br>bcs<br>bcc<br>bvs<br>bvc<br>blt.s<br>bge.s<br>ble.s<br>bgt.s<br>ble.u<br>bgt.u | $cond4, offset[21:1]s<br>(PC relative) | | 1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1 | 16<br>16<br>32<br>32<br>16<br>16<br>32<br>32<br>16<br>16<br>32<br>32<br>32<br>32 |
| 23 | 01 0111 | seq.i<br>sne.i<br>smi.i<br>spl.i<br>scs.i<br>scc.i<br>svs.i<br>svc.i<br>slt.s.i<br>sge.s.i<br>sle.s.i<br>sgt.s.i<br>sle.u.i<br>sgt.u.i | Rd, Rs, $cond4,<br>immediate[12:0]s | | 1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1 | 32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32 |
| 24 | 01 1000 | bra.a | address[25:1]u (Absolute) | | 1 | 32 |
| 25 | 01 1001 | bsr.a | address[25:1]u (Absolute) | | 1 | 32 |
| 26 | 01 1010 | bra.p | offset[25:1]s (PC relative) | | 1 | 16 |

| *OP* | *OP binary* | *6-bit Mnemonic* | *25-bit Argument* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 27 | 01 1011 | bsr.p | offset[25:1]s (PC relative) | | 1 | 32 |
| 28 | 01 1100 | ldm<br>ldm.u | #mask16, @(offset[6:2]u, Ra)<br>Copy low memory addresses to low registers.<br>ldm = Current mode.<br>ldm.u = User mode<br>R0 interpreted as FLAGS in ldm.u. | Z, N<br>On highest reg | n<br>n | 16<br>32 |
| 29 | 01 1101 | ldm.x | #mask16, @(offset[6:2]u, Ra)<br>Copy low memory addresses to high registers. | Z, N<br>On lowest reg | n | 32 |
| 30 | 01 1110 | stm<br>stm.u | #mask16, @(offset[6:2]u, Ra)<br>Copy low registers to low memory addresses.<br>stm = Current mode.<br>stm.u = User mode.<br>R0 interpreted as FLAGS in stm.u. | | n<br>n | 16<br>32 |
| 31 | 01 1111 | stm.x | #mask16, @(offset[6:2]u, Ra)<br>Copy high registers to low memory addresses. | | n | 32 |
| 32 | 10 0000 | add.h | Rd, Rs, #immediate[31:15]h | Z, N, C, V | 1 | 32 |
| 33 | 10 0001 | sub.x.h | Rd, Rs, #immediate[31:15]h | Z, N, C, V | 1 | 32 |
| 34 | 10 0010 | cmp.h<br>cmp.x.h<br>cmp.c.h<br>cmp.xc.h<br>cmp.16.h<br>cmp.16x.h<br>cmp.16c.h<br>cmp.16xc.h | $size2, $order1, $carry1, Rs, #immediate[31:15]h<br><br><br><br>$size2, $order1, $carry1, Rs, #immediate[31:16] | Z, N, C, V | 1<br>1<br>1<br>1<br>1<br>1<br>1<br>1 | 32<br>32<br>32<br>32<br>32<br>32<br>32<br>32 |
| 35 | 10 0011 | and.h | Rd, Rs, #immediate[31:15]h | Z, N | 1 | 32 |
| 36 | 10 0100 | or.h | Rd, Rs, #immediate[31:15]h | Z, N | 1 | 32 |
| 37 | 10 0101 | xor.h | Rd, Rs, #immediate[31:15]h | Z, N | 1 | 32 |
| 38 | 10 0110 | mult.16s.i | Rd, Rs, #immediate[15:0] | Z, N | 1 | 32 |
| 39 | 10 0111 | mult.16u.i | Rd, Rs, #immediate[15:0] | Z, N | 1 | 32 |
| 40 | 10 1000 | add.i | Rd, Rs, #immediate[16:0]s | Z, N, C, V | 1 | 16 |
| 41 | 10 1001 | sub.x.i | Rd, Rs, #immediate[16:0]s | Z, N, C, V | 1 | 16 |

| *OP* | *OP binary* | *6-bit Mnemonic* | *25-bit Argument* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 42 | 10 1010 | cmp.i<br>cmp.x.i<br>cmp.c.i<br>cmp.xc.i<br>cmp.16.i<br>cmp.16x.i<br>cmp.16c.i<br>cmp.16xc.i<br>cmp.8.i<br>cmp.8x.i<br>cmp.8c.i<br>cmp.8xc.i | $size2, $order1, $carry1, Rs, #immediate[16:0]s<br><br><br><br>$size2, $order1, $carry1, Rs, #immediate[15:0]<br><br><br><br>$size2, $order1, $carry1, Rs, #immediate[7:0] | Z, N, C, V | 1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1<br>1 | 16<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32<br>32 |
| 43 | 10 1011 | and.i | Rd, Rs, #immediate[16:0]s | Z, N | 1 | 16 = rlinm |
| 44 | 10 1100 | or.i | Rd, Rs, #immediate[16:0]s | Z, N | 1 | 16 = mov.i |
| 45 | 10 1101 | xor.i | Rd, Rs, #immediate[16:0]s | Z, N | 1 | 16 |
| 46 | 10 1110 | mult.i | Rd, Rs, #immediate[16:0]s | Z, N, C, V | 3 | 32 |
| 47 | 10 1111 | rlinm.s<br>rlinm.z | Rotate Left Immediate, then aNd with Mask $signext1, Rd, Rs, #shift[4:0]u, #highbit[4:0]u, #lowbit[4:0]u | Z, N, C | 1<br>1 | 16<br>16 |
| 48 | 11 0000 | ld.i | Rd, @(offset[16:0]s, Ra) | Z, N | 1 | 16 |
| 49 | 11 0001 | ld.16z.i | Rd, @(offset[16:0]s, Ra) | Z, N | 1 | 16 |
| 50 | 11 0010 | ld.8z.i | Rd, @(offset[16:0]s, Ra) | Z, N | 1 | 16 |
| 51 | 11 0011 | ld.8s.i | Rd, @(offset[16:0]s, Ra) | Z, N | 1 | 32 |
| 52 | 11 0100 | st.i | Rs, @(offset[16:0]s, Ra) | | 1 | 16 |
| 53 | 11 0101 | st.16.i | Rs, @(offset[16:0]s, Ra) | | 1 | 16 |
| 54 | 11 0110 | st.8.i | Rs, @(offset[16:0]s, Ra) | | 1 | 16 |
| 55 | 11 0111 | ld.16s.i | Rd, @(offset[16:0]s, Ra) | Z, N | 1 | 32 |
| 56 | 11 1000 | | | | | |
| 57 | 11 1001 | | | | | |
| 58 | 11 1010 | | | | | |
| 59 | 11 1011 | | | | | |
| 60 | 11 1100 | | | | | |
| 61 | 11 1101 | | | | | |
| 62 | 11 1110 | | | | | |
| 63 | 11 1111 | Extended Opcode | See following table. 16 different OPA[3:0]. | | | |

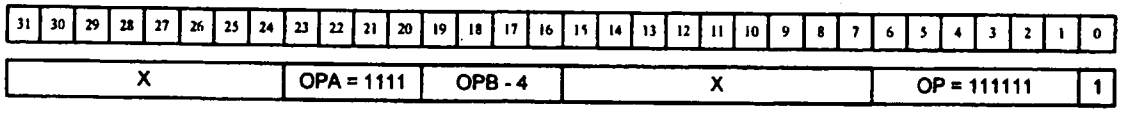
There are 7 primary opcodes free.

### 8.8.2 Extended Opcode – OPA[3:0]



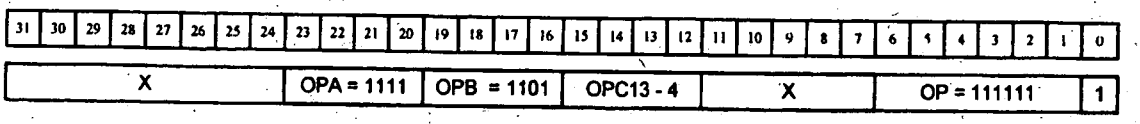

| *OPA* | *OPA binary* | *4-bit Mnemonic* | *8-bit + 13-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | OPE | See following table. 16 different OPE[3:0]. | | | |
| 1 | 0001 | flip.r<br>flip.x.r<br>flip.16x.r<br>flip.8x.r<br>flip.b.r<br>flip.16b.r | $mode3, Rd, Rs, Rt, #lowbyte2 | Z, N | 2<br>2<br>2<br>2<br>2<br>2 | 32<br>32<br>32<br>32<br>32<br>32 |
| 2 | 0010 | mult.sh.r | Rd, Rs, Rt, #shift[5:0]u | Z, N, C, V | 5 | 32 |
| 3 | 0011 | | | | | |
| 4 | 0100 | | | | | |
| 5 | 0101 | | | | | |
| 6 | 0110 | | | | | |
| 7 | 0111 | | | | | |
| 8 | 1000 | | | | | |
| 9 | 1001 | | | | | |
| 10 | 1010 | | | | | |
| 11 | 1011 | | | | | |
| 12 | 1100 | | | | | |
| 13 | 1101 | | | | | |
| 14 | 1110 | | | | | |
| 15 | 1111 | OPB | See following table. 16 different OPB[3:0]. | | | |

### 8.8.3 Extended Opcode - OPB[3:0]



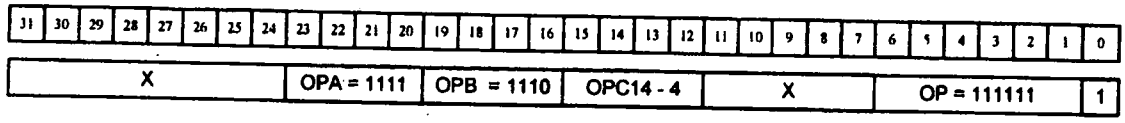

| *OPB* | *OPB binary* | *4-bit Mnemonic* | *8-bit + 9-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | seq.r | $cond4, Rd, Rs, Rt | | 1 | 32 |
| | | sne.r | | | 1 | 32 |
| | | smi.r | | | 1 | 32 |
| | | spl.r | | | 1 | 32 |
| | | scs.r | | | 1 | 32 |
| | | scc.r | | | 1 | 32 |
| | | svs.r | | | 1 | 32 |
| | | svc.r | | | 1 | 32 |
| | | slt.s.r | | | 1 | 32 |
| | | sge.s.r | | | 1 | 32 |
| | | sle.s.r | | | 1 | 32 |
| | | sgt.s.r | | | 1 | 32 |
| | | sle.u.r | | | 1 | 32 |
| | | sgt.u.r | | | 1 | 32 |
| 1 | 0001 | | | | | |
| 2 | 0010 | | | | | |
| 3 | 0011 | | | | | |
| 4 | 0100 | | | | | |
| 5 | 0101 | | | | | |
| 6 | 0110 | | | | | |
| 7 | 0111 | | | | | |
| 8 | 1000 | | | | | |
| 9 | 1001 | | | | | |
| 10 | 1010 | | | | | |
| 11 | 1011 | | | | | |
| 12 | 1100 | | | | | |
| 13 | 1101 | OPC13 | See following table. 16 different OPC13[3:0]. | | | |
| 14 | 1110 | OPC14 | See following table. 16 different OPC14[3:0]. | | | |
| 15 | 1111 | OPC | See following table. 16 different OPC[3:0]. | | | |

### 8.8.4 Extended Opcode – OPC13[3:0]



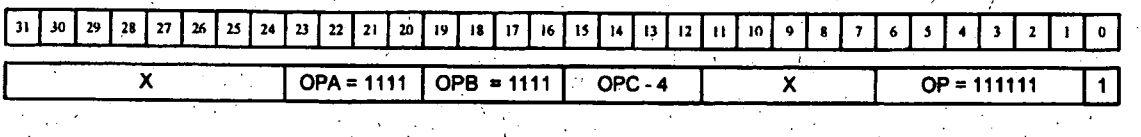

| *OPC 13* | *OPC13 binary* | *4-bit Mnemonic* | *8-bit + 5-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | | | | | |
| 1 | 0001 | | | | | |
| 2 | 0010 | | | | | |
| 3 | 0011 | | | | | |
| 4 | 0100 | | | | | |
| 5 | 0101 | | | | | |
| 6 | 0110 | | | | | |
| 7 | 0111 | | | | | |
| 8 | 1000 | | | | | |
| 9 | 1001 | | | | | |
| 10 | 1010 | | | | | |
| 11 | 1011 | | | | | |
| 12 | 1100 | | | | | |
| 13 | 1101 | | | | | |
| 14 | 1110 | | | | | |
| 15 | 1111 | | | | | |

### 8.8.5 Extended Opcode – OPC14[3:0]



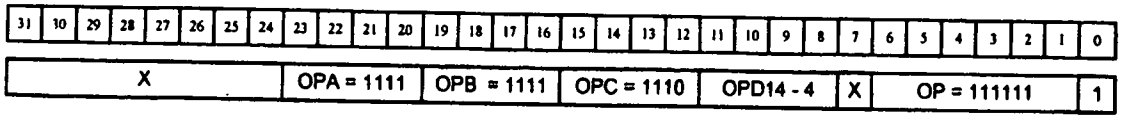

| *OPC 14* | *OPC14 binary* | *4-bit Mnemonic* | *8-bit + 5-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | ld.r | Rd, @(Rx, Ra) | Z, N | 1 | 16 |
| 1 | 0001 | ld.16z.r | Rd, @(Rx, Ra) | Z, N | 1 | 16 |
| 2 | 0010 | ld.8z.r | Rd, @(Rx, Ra) | Z, N | 1 | 16 |
| 3 | 0011 | ld.8s.r | Rd, @(Rx, Ra) | Z, N | 1 | 32 |
| 4 | 0100 | st.r | Rs, @(Rx, Ra) | | 1 | 16 |
| 5 | 0101 | st.16.r | Rs, @(Rx, Ra) | | 1 | 16 |
| 6 | 0110 | st.8.r | Rs, @(Rx, Ra) | | 1 | 16 |
| 7 | 0111 | ld.16s.r | Rd, @(Rx, Ra) | Z, N | 1 | 32 |
| 8 | 1000 | shiftr.s.r | Rd, Rs, Rt | Z, N, C | 1 | 32 |
| 9 | 1001 | shiftr.u.r | Rd, Rs, Rt | Z, N, C | 1 | 32 |
| 10 | 1010 | shiftl.r | Rd, Rs, Rt | Z, N, C | 1 | 32 |
| 11 | 1011 | rotatel.r | Rd, Rs, Rt | Z, N, C | 1 | 32 |
| 12 | 1100 | swap.r | Rd, @(Rx, Ra) | Z, N<br>Set on new Rd value. | 2 | 32 |
| 13 | 1101 | | | | | |
| 14 | 1110 | | | | | |
| 15 | 1111 | | | | | |

### 8.8.6 Extended Opcode – OPC[3:0]



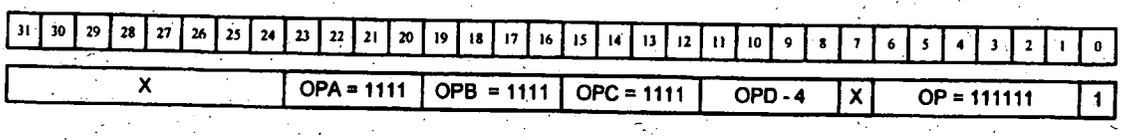

| *OPC* | *OPC binary* | *4-bit Mnemonic* | *8-bit + 5-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | add.r<br>add.c.r | $carry1, Rd, Rs, Rt | Z, N, C, V | 1<br>1 | 16<br>32 |
| 1 | 0001 | sub.r<br>sub.c.r | $carry1, Rd, Rs, Rt | Z, N, C, V | 1<br>1 | 16<br>32 |
| 2 | 0010 | cmp.r<br>cmp.c.r<br>cmp.16.r<br>cmp.16c.r<br>cmp.8.r<br>cmp.8c.r | $size2, $carry1, Rs, Rt | Z, N, C, V | 1<br>1<br>1<br>1<br>1<br>1 | 16<br>32<br>32<br>32<br>32<br>32 |
| 3 | 0011 | and.r | Rd, Rs, Rt | Z, N | 1 | 16 |
| 4 | 0100 | or.r | Rd, Rs, Rt | Z, N | 1 | 16 |
| 5 | 0101 | xor.r | Rd, Rs, Rt | Z, N | 1 | 16 |
| 6 | 0110 | mult.16s.r | Rd, Rs, Rt | Z, N | 1 | 32 |
| 7 | 0111 | mult.16u.r | Rd, Rs, Rt | Z, N | 1 | 32 |
| 8 | 1000 | mult.r | Rd, Rs, Rt | Z, N, C, V | 3 | 16 |
| 9 | 1001 | | | | | |
| 10 | 1010 | div.s.r | Rd, Rs, Rt | Z, N, V | 32 | 32 |
| 11 | 1011 | div.u.r | Rd, Rs, Rt | Z, N, C | 32 | 32 |
| 12 | 1100 | rem.s.r | Rd, Rs, Rt | Z, N | 32 | 32 |
| 13 | 1101 | rem.u.r | Rd, Rs, Rt | Z, N | 32 | 32 |
| 14 | 1110 | OPD14 | See following table. 16 different OPD14[3:0]. | | | |
| 15 | 1111 | OPD | See following table. 16 different OPD[3:0]. | | | |

### 8.8.7 Extended Opcode – OPD14[3:0]



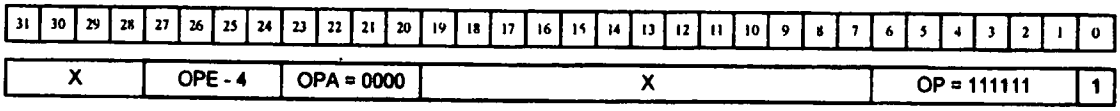

| *OPD 14* | *OPD14 binary* | *4-bit Mnemonic* | *8-bit + 1-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | movr2s | Rd, Rs | | 1 | 32 |
| 1 | 0001 | movs2r | Rd, Rs | Z, N | 1 | 32 |
| 2 | 0010 | movr2b | Rd, Rs | | 1 | 32 |
| 3 | 0011 | movb2r | Rd, Rs | Z, N | 1 | 32 |
| 4 | 0100 | irqe | | | 1 | 32 |
| 5 | 0101 | irqd | | | 1 | 32 |
| 6 | 0110 | irqs | Rd | Z, N | 1 | 32 |
| 7 | 0111 | acc.64.r | Rd, Rs | | 2 | 32 |
| 8 | 1000 | | | | | |
| 9 | 1001 | | | | | |
| 10 | 1010 | msbit.r | Rd, Rs | | 1 | 32 |
| 11 | 1011 | abs.r | Rd, Rs | | 1 | 32 |
| 12 | 1100 | bra.r | Ra | | 1 | 16 |
| 13 | 1101 | bsr.r | Ra | | 1 | 16 |
| 14 | 1110 | trap.r | Rs | | 1 | 16 |
| 15 | 1111 | print.r | Rs | | 1 | 32 |

### 8.8.8 Extended Opcode – OPD[3:0]




| *OPD* | *OPD binary* | *4-bit Mnemonic* | *8-bit + 1-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | | | | | |
| 1 | 0001 | | | | | |
| 2 | 0010 | | | | | |
| 3 | 0011 | | | | | |
| 4 | 0100 | | | | | |
| 5 | 0101 | | | | | |
| 6 | 0110 | | | | | |
| 7 | 0111 | ver | Rd | | | |
| 8 | 1000 | lic | Rd | | | |
| 9 | 1001 | rtie | | Z, N, C, V | 1 | 32 |
| 10 | 1010 | brk | | | 1 | 16 |
| 11 | 1011 | halt | | | 1 | 32 |
| 12 | 1100 | sleepnop | | | 1 | 32 |
| 13 | 1101 | sleepsif | | | 1 | 32 |
| 14 | 1110 | nop | | | 1 | 16 |
| 15 | 1111 | sif | | | 1 | 16 |

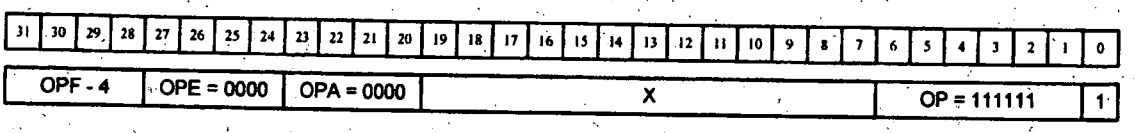
### 8.8.9 Extended Opcode – OPE[3:0]




| *OPE* | *OPE binary* | *4-bit Mnemonic* | *4-bit + 13-bit Arguments* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | OPF | See following table. 16 different OPF[3:0]. | | | |
| 1 | 0001 | bcase.2 | Rs, #immediate[12:0]s | | 3 | 32 |
| 2 | 0010 | bcase.4 | Rs, #immediate[12:0]s | | 3 | 32 |
| 3 | 0011 | | | | | |
| 4 | 0100 | shiftr.64s.i | Rd, #shift[5:0]u | * | 2 | 32 |
| 5 | 0101 | shiftl.64.i | Rd, #shift[5:0]u | * | 2 | 32 |
| 6 | 0110 | shiftr.64s.r | Rd, Rt | * | 2 | 32 |
| 7 | 0111 | shiftl.64.r | Rd, Rt | * | 2 | 32 |
| 8 | 1000 | | | | | |
| 9 | 1001 | | | | | |
| 10 | 1010 | | | | | |
| 11 | 1011 | | | | | |
| 12 | 1100 | | | | | |
| 13 | 1101 | | | | | |
| 14 | 1110 | | | | | |
| 15 | 1111 | | | | | |

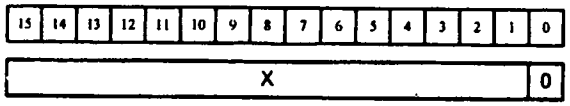
(*) – Note that the 64-bit shifts do not set any flags.

### 8.8.10 Extended Opcode – OPF[3:0]




| *OPF* | *OPF binary* | *4-bit Mnemonic* | *13-bit Argument* | *Flags Updated on Destination. Rd or Memory* | *Cycles* | *Min OP* |
|---|---|---|---|---|---|---|
| 0 | 0000 | trap.i | #immediate[12:0]s | | 1 | 32 |
| 1 | 0001 | | | | | |
| 2 | 0010 | | | | | |
| 3 | 0011 | | | | | |
| 4 | 0100 | | | | | |
| 5 | 0101 | | | | | |
| 6 | 0110 | | | | | |
| 7 | 0111 | | | | | |
| 8 | 1000 | | | | | |
| 9 | 1001 | | | | | |
| 10 | 1010 | | | | | |
| 11 | 1011 | | | | | |
| 12 | 1100 | | | | | |
| 13 | 1101 | | | | | |
| 14 | 1110 | | | | | |
| 15 | 1111 | | | | | |

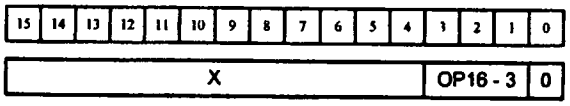
## 8.9 16-bit Instruction Opcodes




Note: the 16-bit instruction 0x0000 is a SIF instruction.

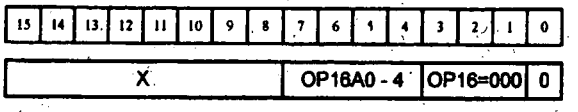
The 16-bit instructions are identical to their 32-bit equivalents. This section describes the relationship between the instruction encodings. For flag information please refer to the relevant 32-bit instruction.

### *8.9.1 16-bit Opcode – OP16[2:0]*



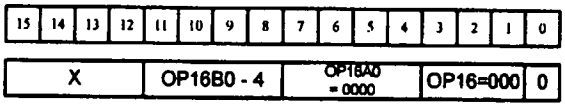

| *OP16* | *OP16 binary* | *3-bit Mnemonic* | *12-bit Argument* |
|---|---|---|---|
| 0 | 000 | OP16A0 | See following table. 16 different OP16A0[3:0]. |
| 1 | 001 | OP16A1 | See following table. 4 different OP16A1[1:0]. |
| 2 | 010 | OP16A2 | See following table. 4 different OP16A2[1:0]. |
| 3 | 011 | ld.i | Rd, @(offset[6:2]u, Ra)<br>Rd: R1-R8 |
| 4 | 100 | st.i | Rs, @(offset[6:2]u, Ra)<br>Rs: R0-R7 |
| 5 | 101 | OP16A5 | See following table. 2 different OP16A5[0]. |
| 6 | 110 | OP16A6 | See following table. 6 different OP16A6[2:0]. |
| 7 | 111 | OP16A7 | See following table. 7 different OP16A7[3:0]. |

### 8.9.2 16-bit Extended Opcode – OP16A0[3:0]



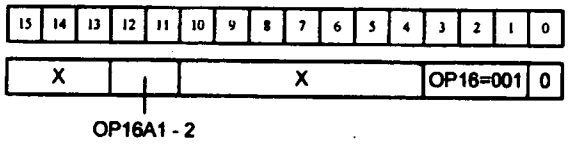

| *OP16A0* | *OP16A0 binary* | *4-bit Mnemonic* | *8-bit Argument* |
|---|---|---|---|
| 0 | 0000 | OP16B0 | See following table. 16 different OP16B0[3:0]. |
| 1 | 0001 | add.r | Rd, Rd, Rt<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 2 | 0010 | sub.r | Rd, Rd, Rt<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 3 | 0011 | add.i | Rd, Rs, #1<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 4 | 0100 | and.r | Rd, Rd, Rt<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 5 | 0101 | or.r | Rd, Rd, Rt<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 6 | 0110 | xor.r | Rd, Rd, Rt<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 7 | 0111 | mult.r | Rd, Rd, Rt<br>argument = 0b0000XXXX is reserved<br>argument = 0bXXXX0000 is reserved |
| 8 | 1000 | ld.r | Rd, @(Rx, Ra) |
| 9 | 1001 | ld.16z.r | Rd, @(Rx, Ra) |
| 10 | 1010 | ld.8z.r | Rd, @(Rx, Ra) |
| 11 | 1011 | mov.r | Rd, Rt (= or.r Rd, %r0, Rt)<br>argument = 0b0000XXXX is reserved |
| 12 | 1100 | st.r | Rs, @(Rx, Ra) |
| 13 | 1101 | st.16.r | Rs, @(Rx, Ra) |
| 14 | 1110 | st.8.r | Rs, @(Rx, Ra) |
| 15 | 1111 | cmp.r | Rs, Rt |

### 8.9.3 16-bit Extended Opcode – OP16B0 [3:0]



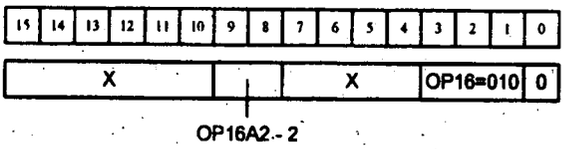

| *OP16B0* | *OP16B0 binary* | *4-bit Mnemonic* | *4-bit Argument* |
|---|---|---|---|
| 0 | 0000 | sif | 0 = sif<br>1 = nop<br>2 = brk |
| 1 | 0001 | xor.i | Rd, Rd, #0xFFFFFFFF (i.e a NOT instruction) |
| 2 | 0010 | sub.x.i | Rd, Rd, #0 (i.e a NEG instruction) |
| 3 | 0011 | | |
| 4 | 0100 | bra.r | Ra |
| 5 | 0101 | bsr.r | Ra |
| 6 | 0110 | | |
| 7 | 0111 | | |
| 8 | 1000 | seq.i | Rd, %r0, #1 |
| 9 | 1001 | sne.i | Rd, %r0, #1 |
| 10 | 1010 | seq.i | Rd, %r0, #-1 |
| 11 | 1011 | sne.i | Rd, %r0, #-1 |
| 12 | 1100 | | |
| 13 | 1101 | | |
| 14 | 1110 | | |
| 15 | 1111 | | |

### 8.9.4 16-bit Extended Opcode – OP16A1[1:0]



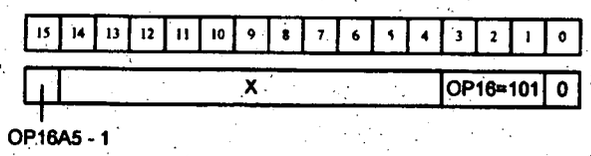

| *OP16A1* | *OP16A1 binary* | *2-bit Mnemonic* | *10-bit Argument* |
|---|---|---|---|
| 0 | 00 | add.i | Rd, Rd, #imm[6:0]s |
| 1 | 01 | add.i | Rd, R15, #imm[8:2]s |
| 2 | 10 | cmp.i | Rs, #imm[6:0]u |
| 3 | 11 | mov.i | Rd, #imm[6:0]u (= or.i Rd, %r0, #imm) |

### 8.9.5 16-Bit Extended Opcode – OP16A2[1:0]



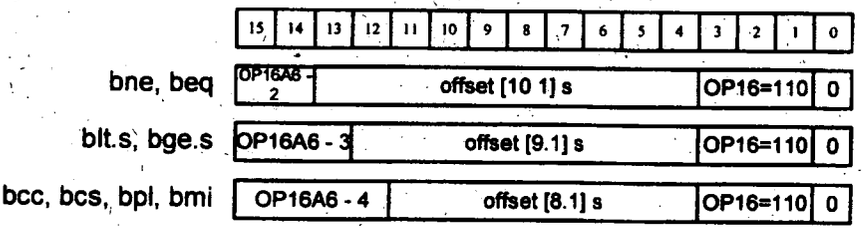

| *OP16A2* | *OP16A2 binary* | *2-bit Mnemonic* | *10-bit Argument* |
|---|---|---|---|
| 0 | 00 | ld.8z.i | Rd, @(offset[3:0]u, Ra) |
| 1 | 01 | ld.16z.i | Rd, @(offset[4:1]u, Ra) |
| 2 | 10 | st.8.i | Rs, @(offset[3:0]u, Ra) |
| 3 | 11 | st.16.i | Rs, @(offset[4:1]u, Ra) |

### 8.9.6 16-bit Extended Opcode – OP16A5




| *OP16A5* | *OP16A5 binary* | *1-bit Mnemonic* | *11-bit Argument* |
|---|---|---|---|
| 0 | 0 | movm | #RdPair3, RsA, RsB |
| 1 | 1 | bra.p | offset[11:1]s |

### 8.9.7 16-bit Extended Opcode – OP16A6[1:0]

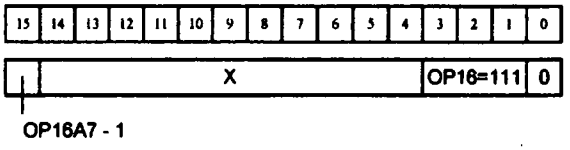



| *OP16A6* | *OP16A6 binary* | *2-bit Mnemonic* | *10-bit Argument* |
|---|---|---|---|
| 0, 1, 2, 3 | 00XX | beq | offset[10:1]s |
| 4, 5, 6, 7 | 01XX | bne | offset[10:1]s |
| 8, 9 | 100X | blt.s | offset[9:1]s |
| 10, 11 | 101X | bge.s | offset[9:1]s |
| 12 | 1100 | bmi | offset[8:1]s |
| 13 | 1101 | bpl | offset[8:1]s |

| *OP16A6* | *OP16A6 binary* | *2-bit Mnemonic* | *10-bit Argument* |
|---|---|---|---|
| 14 | 1110 | bcs | offset[8:1]s |
| 15 | 1111 | bcc | offset[8:1]s |

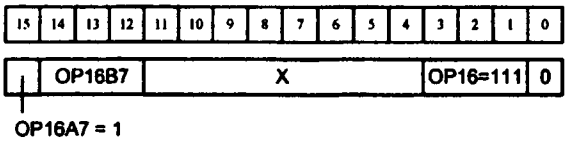
In all cases if the branch condition is met then PC = PC + offset.

### 8.9.8 16-bit Extended Opcode – OP16A7




| *OP16A7* | *OP16A7 binary* | *1-bit Mnemonic* | *11-bit Argument* |
|---|---|---|---|
| 0 | 0 | rlinm.s<br>rlinm.z | #mode2, #sign1, Rd, #imm[4:0]u |
| 1 | 1 | OP16B7 | See following table for OP16B7[2:0]. |

### 8.9.9 16-bit Extended Opcode – OP16B7




| *OP16B7* | *OP16B7 binary* | *3-bit Mnemonic* | *8 or 9-bit Argument* |
|---|---|---|---|
| 0, 1 | 00X | ldm | Ra , #RegMask[5:1] |
| 2, 3 | 01X | stm | Ra , #RegMask[5:1] |
| 4, 5 | 10X | push, push.r | #RegMask[5:1], #RegSet4 |
| 6 | 110 | pop | #retval2, #Offset[3:2]u, #RegSet4 |
| 7 | 111 | pop.ret | #retval2, #Offset[3:2]u, #RegSet4 |

## 8.10 32-Bit Instruction Formats

### *8.10.1 Unconditional Branch Instructions*

bra.a, bra.p, bra.r

bsr.a, bsr.p, bsr.r

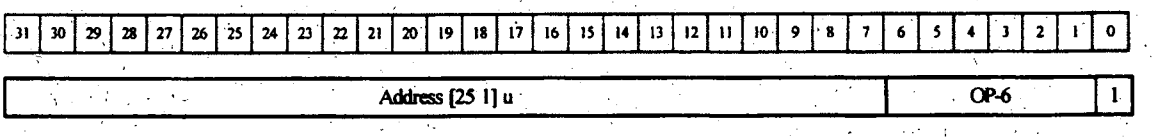
Two unconditional branches are provided :

- bra (Branch) will change the program counter to the given argument.
- bsr (Branch Subroutine) will change the program counter and save the address of the instruction after the branch into the link register, R14.

The bra.* and bsr.* instructions do not update any flags.

#### 8.10.1.1 Absolute

As instructions must start on a 16-bit word boundary the least significant bit (Address[0]) of the target is assumed to be 0, and appended to Address[25:1] given in the instruction. The 25-bit value in the instruction thus covers the lower 26-bits of the address space.

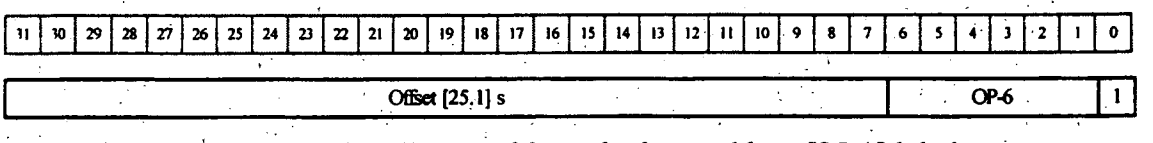



```
bra.a   label          // Assembler calculates address[25:1] label
        bra.a  label129  // Assembler calculates address[25:1] label129
        bsr.a  label35   // Assembler calculates address[25:1] label35
```

#### 8.10.1.2 PC Relative

This branches to an offset address from the Program Counter.

As with the absolute mode, bit 0 of the offset is assumed to be 0.




```
bra.p   label          // Assembler calculates address[25:1] label
        bra.p  label32   // Assembler calculates address[25:1] label32
        bsr.p  label782  // Assembler calculates address[25:1] label782
```

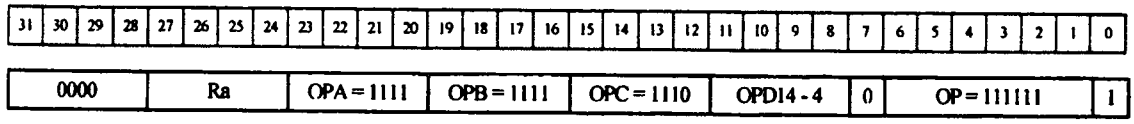
Sometimes the compiler wants to force the bra.p to be 2byte or 4byte instructions. In these cases it will generate the mnemonics :

```
bra.p.2         // 2byte, 16-bit instruction
```

bra.p.4 // 4byte, 32-bit instruction

### 8.10.1.3 Register

The new value of the program counter is taken from the indicated register. Unlike the absolute and relative addressing modes, no address shift is needed.)




bra.r Ra // No flags set

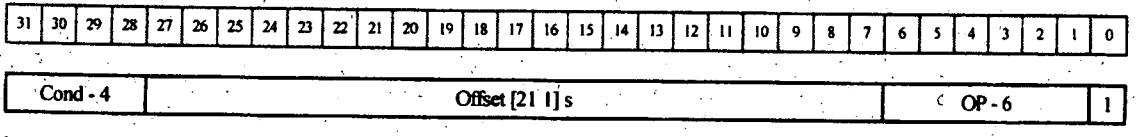
bra.r %r4

bsr.r %r12

### 8.10.2 Conditional Branch Instructions

All conditional branches are relative to the Program Counter. All instructions start on a 16-bit boundary. Bit 0 of the offset is thus assumed to be 0. The 21-bit signed offset allows functions to be up to 2 MB.

There are 14 conditional branches.




```
beq     label           //BinUtils will calculate offset[21:1] from this label
        beq     label782    // BinUtils will calculate offset[21:1] from label782
        blt.s   labelA      // BinUtils will calculate offset[21:1] from labelA
```

The conditional branches will occur when the flags are set as shown in the following tables :

| *Cond* | *Mnemonic* | *Branch If ...* | *Flag Values Required for a Branch* |
|---|---|---|---|
| 0 | beq | Equal (Zero Set) | Z == 1 |
| 1 | bne | Not Equal (Zero Clear) | Z == 0 |
| 2 | bmi | Minus (Negative Set) | N == 1 |
| 3 | bpl | Plus (Negative Clear) | N == 0 |
| 4 | bcs (= blt.u) | Carry Set | C == 1 |
| 5 | bcc (= bge.u) | Carry Clear | C == 0 |
| 6 | bvs | Overflow Set | V == 1 |
| 7 | bvc | Overflow Clear | V == 0 |
| 8 | blt.s | Less Than (signed) | N != V |
| 9 | bge.s | Greater Than or Equal (signed) | N == V |
| 10 | ble.s | Less Than or Equal (signed) | (N != V) \|\| (Z == 1) |
| 11 | bgt.s | Greater Than (signed) | (N == V) && (Z == 0) |
| 12 | ble.u | Less Than or Equal (unsigned) | (C == 1) \|\| (Z == 1) |
| 13 | bgt.u | Greater Than (unsigned) | (C == 0) && (Z == 0) |
| 14 | | | |
| 15 | | | |

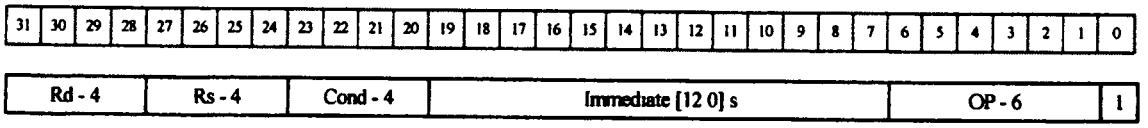
### 8.10.3 Conditional Set

There are register and immediate forms of the conditional set instruction.

The immediate form tests the flags for a condition and sets the destination register, Rd, to the source register, Rs, if the condition is met or to the immediate value otherwise.

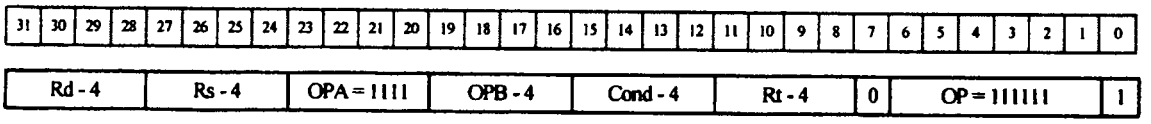
The register form of the instruction sets the destination to register to the value of one of the two specified source registers. If the condition is met Rd becomes equal to Rs, otherwise Rd becomes equal to Rt. By setting one of the source registers to be the same as the destination register, this instruction can be used to conditionally move a register.

The same set of conditions is used in these instructions as for the conditional branches. The table in the previous section describes the conditions.




seq.i Rd, Rs, #imm // Rd is set to Rs if condition met, else to imm

seq.i %r4, %r0, #15




seq.r Rd, Rs, Rt // Rd is set to Rs if condition met otherwise to Rt

seq.r %r4, %r7, %r8

These instructions are useful for the C ternary operator.

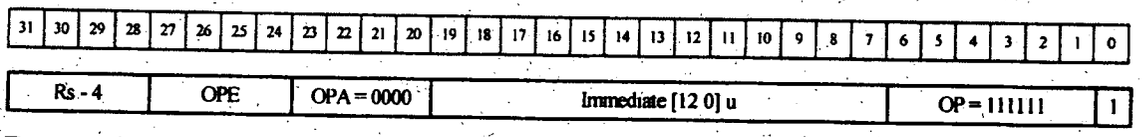
### 8.10.4 Case Branch Instructions

The bcase.2 and bcase.4 instructions test whether a register is less than a specified immediate and conditionally branch to a destination calculated from the register. These instructions are similar to conditional branches, except that they have a variable branch destination.

These instructions are intended to be used to implement C switch-case statements. In a switch statement implementation, the bcase instructions test whether the switch variable is less than the number of case values and sets the PC to a position in a jump table according to the switch variable. The default case is not to take a branch. Therefore, the jump table starts with the default case.

Bcase.4 is used with jump tables consisting of 4 byte branch instructions.

Bcase.2 is used with jump tables consisting of 2 byte branch instructions.




Bcase.4 is implemented as:

```
PC = next instruction
cmp Rs, Imm
if Rs < Imm then
      PC += 4 * (Rs + 1)
endif
```

Bcase.2 is implemented as:

```
PC = next instruction
cmp Rs, Imm
if Rs < Imm then
      PC += 2 * (Rs + 1)
endif
```

bcase.2 Rs, #immediate13u

bcase.4 Rs, #immediate13u

### 8.10.5 Load and Store Instructions

ld.i, ld.16s.i, ld.16z.i, ld.8s.i, ld.8z.i // Displacement Loads

ld.r, ld.16s.r, ld.16z.r, ld.8s.r, ld.8z.r // Indexed Loads

st.i, st.16.i, st.8.i // Displacement Stores

st.r, st.16.r, st.8.r // Indexed Stores

The ld instructions allow data to be copied from memory (or memory mapped registers) to a specified register (Rd). If Rd is R0, then the memory will be read, but the contents will not be written to any register (as R0 is a constant 0).

The st instructions allow data to be copied from a specified register (Rs) to memory.

For 8/16-bit loads to register, the higher bits of the register are set to 0 or sign extended according to whether the instruction is a '.z' or '.s' form.

All the ALU operations operate on complete 32-bit registers. The ld and st instructions support 32, 16 and 8-bit operations. It is these instructions that enable the XAP3 to support char and short data types.

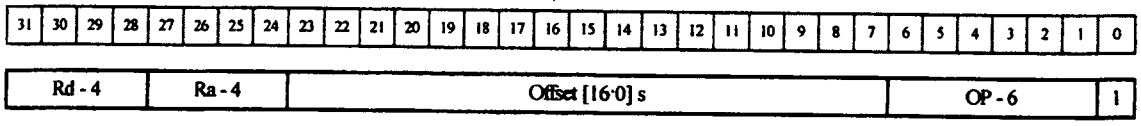
The only XAP3 instructions that can write to memory are:

- st.i, st.16.i, st.8.i, st.r, st.16.r, st.8.r
- stm, stm.u, stm.x
- push, push.r
- swap.r

#### 8.10.5.1 Displacement Addressing (offset, Ra) – ld.*.i, st.*.i

Displacement Load and Store instructions are available for 32, 16 and 8-bit data. 16 and 8-bit data is aligned to bit 0 (in register or memory).

Offset and Ra are both interpreted as Byte addresses.




ld.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra)

ld.16s.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra), sign extend

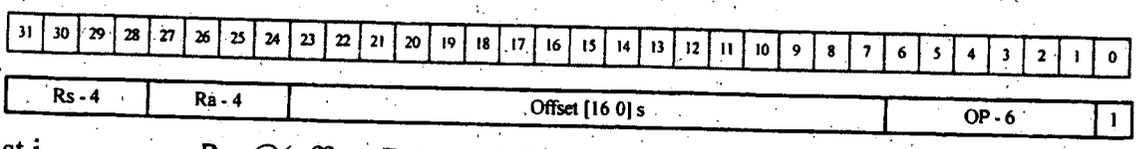
ld.16z.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra), zero extend

ld.8s.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra), sign extend

ld.8z.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra), zero extend

```
ld.i    %r1, @(0x35d, %r14)
```

```
ld.16z.i    %r1, @(27, %r12)
ld.8s.i     %r1, @(357, %r0)    // Forms the absolute address 357.
```




```
st.i        Rs, @(offset, Ra)    // Store Rs to address (offset + Ra)
st.16.i     Rs, @(offset, Ra)    // Store Rs to address (offset + Ra)
st.8.i      Rs, @(offset, Ra)    // Store Rs to address (offset + Ra)

    st.i        %r1, @(0x35d, %r14)
    st.16.i     %r1, @(27, %r12)
    st.8.i      %r1, @(357, %r0)    // Forms the absolute address 357.
```

When using the 8-bit or 16-bit instructions (ld.8s.i, ld.8z.i, st.8.i, ld.16s.i, ld.16z.i, st.16.i) the requested size is indicated to the xap3_mmu. For example, accessing a 16-bit memory with ld.16s.i should result in only one memory access rather than the two required for a 32-bit load.

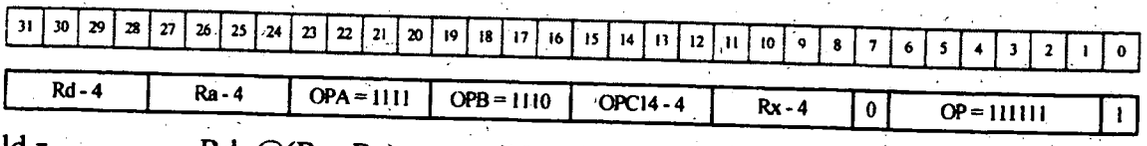
### 8.10.5.2 Indexed Addressing (Rx, Ra) – ld.*.r, st.*.r, swap.r

Indexed Load and Store instructions are available for 32, 16 and 8-bit data. 16 and 8-bit data is aligned to bit 0 (in register or memory).

Ra is interpreted as a Byte address. Rx is scaled according to the width of the data being loaded or stored:

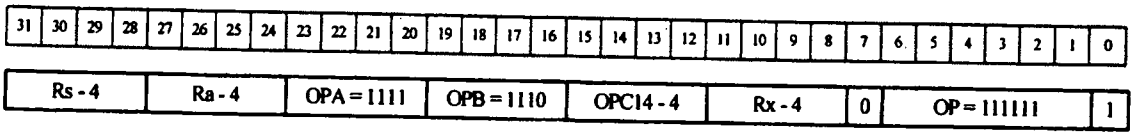
- Address = n*Rx + Ra
- ld.r, st.r — Rx is multiplied by 4
- ld.16s.r, ld.16z.r, st.16.r — Rx is multiplied by 2
- ld.8s.r, ld.8z.r, st.8.r — Rx is not scaled

When using the 8-bit or 16-bit instructions (ld.8s.r, ld.8z.r, st.8.r, ld.16s.r, ld.16z.r, st.16.r) the requested size is indicated to the xap3_mmu. For example, accessing a 16-bit memory with ld.16z.r should result in only one memory access rather than the two required for a 32-bit load.




```
ld.r        Rd, @(Rx, Ra)       // Load Rd from address (4*Rx + Ra)
ld.16s.r    Rd, @(Rx, Ra)       // Load Rd from address (2*Rx + Ra), sign extend
```

```
ld.16z.r    Rd, @(Rx, Ra)       // Load Rd from address (2*Rx + Ra), zero extend
ld.8s.r     Rd, @(Rx, Ra)       // Load Rd from address (Rx + Ra), sign extend
ld.8z.r     Rd, @(Rx, Ra)       // Load Rd from address (Rx + Ra), zero extend
        ld.r        %r1, @(%r14, %r3)   // address = 4*R14 + R3
        ld.16z.r    %r1, @(%r3, %r12)   // address = 2*R3 + R12
        ld.8s.r     %r1, @(%r0, %r10)   // address = R10
```




```
st.r        Rs, @(Rx, Ra)       // Store Rs to address (4*Rx + Ra)
st.16.r     Rs, @(Rx, Ra)       // Store Rs to address (2*Rx + Ra)
st.8.r      Rs, @(Rx, Ra)       // Store Rs to address (Rx + Ra)
        st.r        %r1, @(%r14, %r3)   // address = 4*R14 + R3
        st.16.r     %r1, @(%r3, %r12)   // address = 2*R3 + R12
        st.8.r      %r1, @(%r0, %r10)   // address = R10
```

#### 8.10.5.2.1 swap.r Instruction

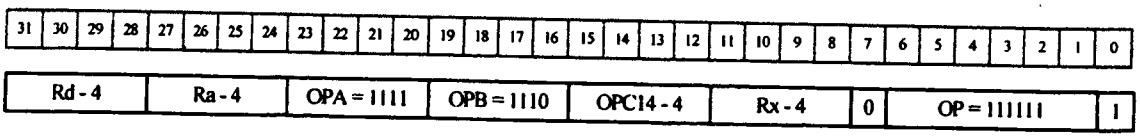
Performs an atomic swap between a register and a memory location.

This instruction asserts a signal from the Core to the xap3_mmu to request that the memory bus is not released. The atomicity of the instruction assumes a compliant xap3_mmu.

```
swap.r Rd, @(Rx, Ra)
```

The memory byte address is calculated as :

- Address = 4*Rx + Ra  // offset is signed. Data is 32-bit only.



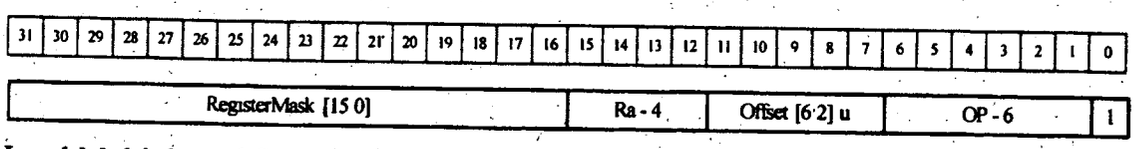

This instruction is not intended to be used by the C compiler.

However, it is a very useful assembler instruction for implementing semaphores. This is because the instruction is atomic and therefore cannot be interrupted half way through execution.

### 8.10.6 Load and Store Multiple Instructions

ldm, ldm.x, ldm.u

stm, stm.x, stm.u




Load Multiple and Store Multiple instructions are provided to simplify context switches and procedure calls. These allow a block of registers to be loaded (from memory) or stored (to memory) with only one instruction. A 16-bit mask indicates which registers (Flags, R1 ... R15) should be loaded or stored. The %r0 bit represents the FLAGS register in the ldm.u and stm.u instructions and %r0 in the others. The ldm.u and stm.u instructions can only be used in the privileged modes.

The order in which ldm copies data from memory to registers, or stm copies data from registers to memory is implementation-defined. Therefore ldm and stm should not be used for accessing volatile or memory-mapped locations where the order of access is important. The instructions are all atomic and therefore cannot be interrupted. What is important is which registers/flags go to/from which addresses. The instructions assume that the memory area used is a contiguous sequence of 32-bit words mapped as follows :

ldm, stm:

| *Memory Address* | *Register* |
|---|---|
| Highest | R15 |
| | R14 |
| | ... |
| | R1 |
| Lowest | R0 |

ldm.u, stm.u:

| *Memory Address* | *Register* |
|---|---|
| Highest | R15 |
| | R14 |
| | ... |
| | R1 |
| Lowest | Flags |

ldm.x, stm.x :

| *MemoryAddress* | *Register* |
|---|---|
| Highest | R0 |
| | R1 |
| | ... |
| | R14 |

| *MemoryAddress* | *Register* |
|---|---|
| Lowest | R15 |

ldm.x is the same as ldm apart from the memory order. If R0 is contained in the register list, the memory will be read, but the contents will not be written to any register (as R0 is a constant 0). The compiler may choose to do this for variables declared as 'volatile' (e.g a FIFO).

stm.x is the same as stm apart from the memory order.

The four major instructions (ldm, ldm.x, stm, stm.x) allow offset[6:2] to be 0 to 30. Offset[6:2] = 31 is used to select the two minor instructions (ldm.u, stm.u). Thus the operands to these two minor instructions must always specify a 0 offset in the source code, which is mapped to the immediate value 31 in the assembler:

| *Opcode* | *Offset[6:2]u* | *Mnemonic* | *Base Address* |
|---|---|---|---|
| 28 | 0 – 30 | ldm | Ra + 4*offset[6:2] |
| 28 | 31 | ldm.u | Ra |
| 29 | 0 – 30 | ldm.x | Ra + 4*offset[6:2] |
| 29 | 31 | Reserved Instruction | |
| 30 | 0 – 30 | stm | Ra + 4*offset[6:2] |
| 30 | 31 | stm.u | Ra |
| 31 | 0 – 30 | stm.x | Ra + 4*offset[6:2] |
| 31 | 31 | Reserved Instruction | |

Note that the indicated registers are stored/loaded sequentially in memory, the address offset is not increased for registers for which the bit is not set in the mask.

When using ldm, it is acceptable for Ra to be one of the registers set in RegisterMask[15:0]. The hardware has a separate temporary register to hold Ra at the start of ldm.

#### 8.10.6.1 Modes

The 4 major instructions (ldm, ldm.x, stm, stm.x) can be used in all 4 modes (User, Supervisor, Interrupt and NMI). They always refer to the current mode.

The 2 minor instructions (ldm.u, stm.u) can only be used in Supervisor and Interrupt modes. Their meanings when the processor is in these modes is as follows:

| *Instruction* | *Current Mode = User* | *Current Mode = Supervisor* | *Current Mode = Interrupt* | *Current Mode = NMI* |
|---|---|---|---|---|
| ldm | Load memory to User Mode Registers | Load memory to Supervisor Mode Registers | Load memory to Interrupt Mode Registers | Load memory to NMI Mode Registers |

| | | | | |
|---|---|---|---|---|
| ldm.x | Load memory to User Mode Registers | Load memory to Supervisor Mode Registers | Load memory to Interrupt Mode Registers | Load memory to NMI Mode Registers |
| ldm.u | IllegalInstructi on | Load memory to User Mode Registers | Load memory to User Mode Registers | Load memory to User Mode Registers |

| *Instruction* | *Current Mode = User* | *Current Mode = Supervisor* | *Current Mode = Interrupt* | *Current Mode = NMI* |
|---|---|---|---|---|
| stm | Store User Registers to Memory | Store Supervisor Registers to Memory | Store Interrupt Registers to Memory | Store NMI Registers to Memory |
| stm.x | Store User Registers to Memory | Store Supervisor Registers to Memory | Store Interrupt Registers to Memory | Store NMI Registers to Memory |
| stm.u | IllegalInstructi on | Store User Registers to Memory | Store User Registers to Memory | Store User Registers to Memory |

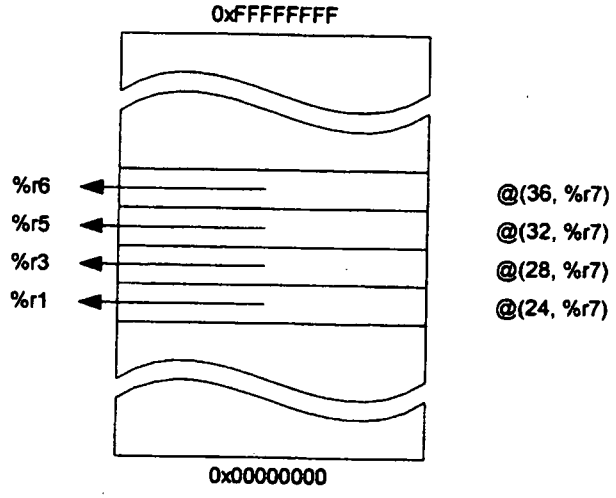
#### 8.10.6.2 ldm.*

The example below is acceptable in Interrupt Mode because the first %r15 is R15_U (in the register list) and the second is R15_I (in the memory address). R15 is the Stack Pointer :

```
        ldm.u {%r14, %r15}, @(0, %r15)     // R15 = SP
ldm     {register list}, @(offset[6:2]u, Ra)   // offset specified in bytes
                                               // Registers in list must be from low to high
```

Here is an example of how ldm is executed :

```
        ldm     {%r1, %r3, %r5, %r6}, @(24, %r7)  // Load current registers
```




Here are examples of other ldm.* instructions :

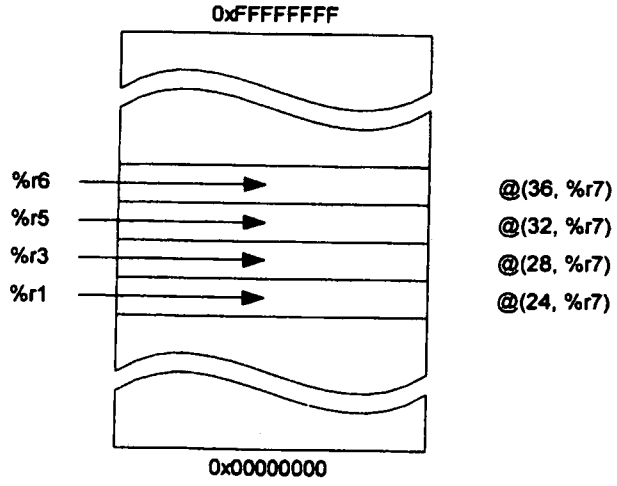
```
ldm.x  {%r1, %r3, %r5, %r6}, @(104, %r7) // Load current registers
ldm.u  {%r1, %r3, %r5, %r6}, @(0, %r7)   // Load User registers
```

### 8.10.6.3 stm.*

```
stm    {register list}, @(offset[6:2]u, Ra)   // offset specified in bytes
                                              // Registers in list must be from low to high
```

Here is an example of how stm is executed :

```
stm    {%r1, %r3, %r5, %r6}, @(24, %r7)  // Store current registers
```




Here are examples of other stm.* instructions :

```
stm.x  {%r1, %r3, %r5, %r6}, @(120, %r7) // Store current registers
stm.u  {%r1, %r5, %r6}, @(0, %r7)        // Store User registers
```

### 8.10.7 push.*, pop.* Instructions

These instructions enable very compact code for stack operations and function entry and exit.

The Stack Pointer (SP) is %r15. The stack grows downwards in memory. i.e SP starts as a high address and becomes a lower address as the stack increases in size. This means that when accessing stack data, stack offsets will always be positive.

So the convention for the stack pointer is :

- Push data. Stack grows. Stack Pointer becomes more negative.
- Pop data. Stack shrinks. Stack Pointer becomes more positive.

The instructions have various modes. All are passed :

- A register list (different registers available for push and pop).
- A stack pointer offset. This is a 10-bit unsigned number of 32-bit words. The instruction increases or decreases the stack pointer (%r15) by this offset in addition to 4n, for the n selected registers. So the stack pointer (byte address) can be changed by 0 to 4092 by each of these instructions, in addition to the increase required for the selected registers.

In all these cases SP (%r15) is increased or decreased by (4n + offset), where n is the number of selected registers. Offset should be interpreted as 4*offset[11:2] (which is an unsigned number).

All the push.* and pop.* instructions specify a fixed relationship between registers and memory :

- Which registers are copied to which memory addresses by push.*
- Which memory addresses are copied to which registers by pop.*

However, they do not specify the order in which the copies should be done. The instructions are all atomic (non-interruptible) and so the copy order should be chosen to ease hardware design.

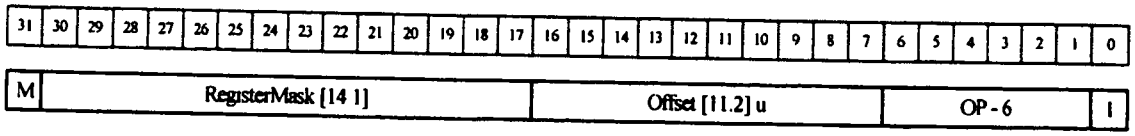
The memory/register relationship used is :

- low registers mapped to low memory addresses (low stack offsets)
- high registers mapped to high memory addresses (high stack offsets)

The normal version of push copies all selected registers to the stack.

The special push.r instruction copies selected R6-R14 to the stack, then selected R1-R5 to higher registers. Therefore this uses less memory accesses than push.

### 8.10.7.1 push, push.r




The register list can include any of R1-R14.

The mode bit M is interpreted as follows :

| *M* | *Mnemonic* | *Meaning* |
|---|---|---|
| 0 | push | Store selected R1-R14 to Stack memory |
| 1 | push.r | Store selected R6-R14 to Stack Memory.<br>Then copy selected R1-R5 to selected R6-R14.<br>Set condition flags on result of last copy. |

Both instructions are identical when R1-R5 are not in the RegisterMask.

#### 8.10.7.1.1 push

The operation sequence is :

- Decrease stack pointer (%r15) by (offset + 4n), where n is the number of selected registers.
- Store selected registers to memory. %r14 first, to the highest address. %r1 last to the lowest address.

The mask is specified as a set of comma-separated registers (low to high) inside curly brackets. The mask cannot convey the order in which the registers will be pushed. The processor will push registers from %r14 first to %r1 last.

The assembler syntax for push.* is :

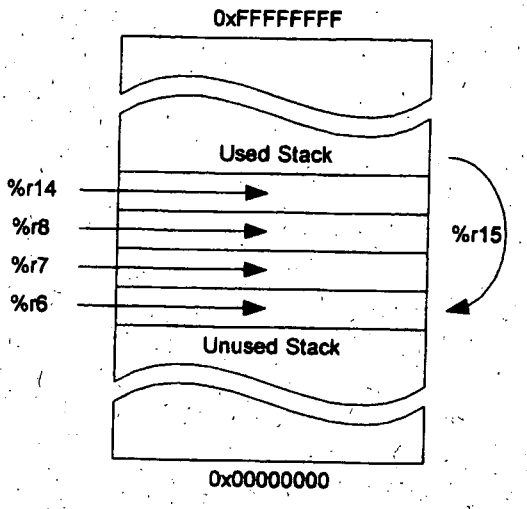
```
push    {register_list}, #offset[11:2]u        // offset specified in bytes
```

##### *8.10.7.1.1.1 push example 1*

```
push    {%r6, %r7, %r8, %r14}, #0   // Registers must be from low to high
```

This is equivalent to the following sequence of instructions.

```
add.i   %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st      %r14, @(0, %r15)     // Store R14 to stack memory
add.i   %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st      %r8, @(0, %r15)      // Store R8 to stack memory
add.i   %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st      %r7, @(0, %r15)      // Store R7 to stack memory
add.i   %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st      %r6, @(0, %r15)      // Store R6 to stack memory
```

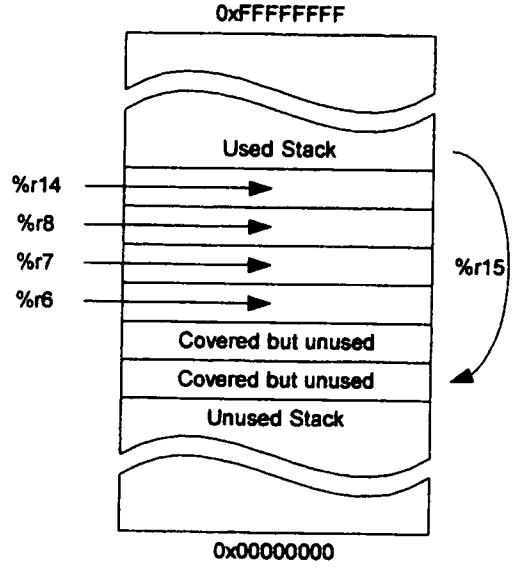



*8.10.7.1.1.2 push example 2*

```
push   {%r6, %r7, %r8, %r14}, #8   // Registers must be from low to high
```

This is equivalent to the following sequence of instructions.

```
add.i  %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st     %r14, @(0, %r15)     // Store R14 to stack memory
add.i  %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st     %r8, @(0, %r15)      // Store R8 to stack memory
add.i  %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st     %r7, @(0, %r15)      // Store R7 to stack memory
add.i  %r15, %r15, #-4      // Decrease Stack Pointer for 1 register
st     %r6, @(0, %r15)      // Store R6 to stack memory
add.i  %r15, %r15, #-8      // Decrease Stack Pointer by #offset[11:2]u
                            // 8 bytes of stack free for other purposes
```




### 8.10.7.1.2 push.r

The push.r register list cannot contain more R1-R5 registers than R6-R14 registers. This is because there must be enough vacated R6-R14 registers (free after being pushed to the stack) to take the R1-R5 registers that need to be moved to R6-R14.

The operation sequence is :

- Decrease stack pointer (%r15) by (offset + 4n), where n is the number of selected registers.
- Store selected R6-R14 registers to memory. %r14 first, to the highest address. %r6 last to the lowest address.
- Copy selected R1-R5 to selected R6-R14. There must not be more registers in the set (R1-R5) than there are in the set (R6-R14).
- Set the condition flags on the result of the last copy operation. If there is no copy (R1-R5 are not in the register set) then the flags are unchanged.

The mask is specified as a set of comma-separated registers inside curly brackets. The register sequence must be from low to high. The mask cannot convey the order in which the registers will be pushed. The processor will push registers from %r14 first to %r6 last.

The assembler syntax for push.r is :

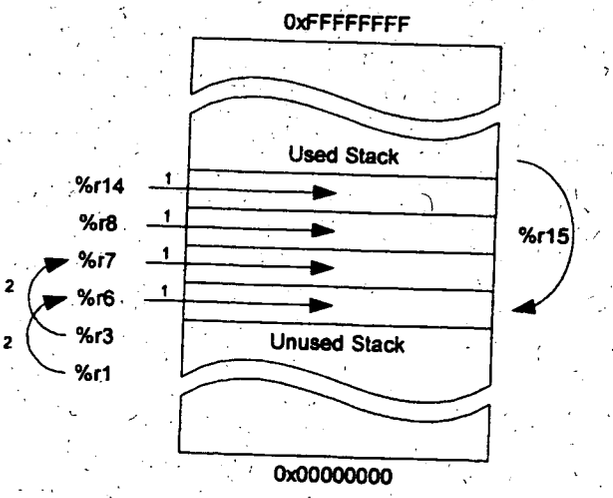
```
push.r  {register_list}, #offset[11:2]u        // offset specified in bytes
```

*8.10.7.1.2.1 push.r example*

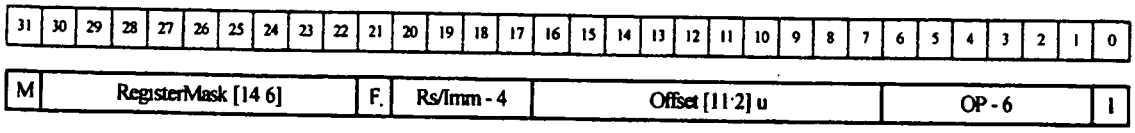
```
push.r {%r1, %r3, %r6, %r7, %r8, %r14}, #0
        // Registers must be from low to high
```

This is equivalent to the following sequence of instructions.

```
add.i   %r15, %r15, #-4     // Decrease Stack Pointer for 1 register
st      %r14, @(0, %r15)    // Store R14 to stack memory
add.i   %r15, %r15, #-4     // Decrease Stack Pointer for 1 register
st      %r8, @(0, %r15)     // Store R8 to stack memory
add.i   %r15, %r15, #-4     // Decrease Stack Pointer for 1 register
st      %r7, @(0, %r15)     // Store R7 to stack memory
add.i   %r15, %r15, #-4     // Decrease Stack Pointer for 1 register
st      %r6, @(0, %r15)     // Store R6 to stack memory
mov.r   %r7, %r3            // Copy R3 to R7 (2nd of R6-R14 in register list)
mov.r   %r6, %r1            // Copy R1 to R6 (1st of R6-R14 in register list)
                            //  and set the Z, N flags accordingly.
```



### 8.10.7.2 pop, pop.ret




The register list can include any of R6-R14. The register sequence (in the assembler) must be from low to high.

Note that pop.* cannot block copy memory to R1-R5 and so ldm must be used instead. This is not normally needed as R1-R5 are not preserved after a function call.

The mode bit, M, encodes whether the instruction is a pop (value 0) or a pop.ret (value 1). The flag bit, F, encodes whether the return value is from a register (value 0) or an immediate (value 1).

The mode bit M and flag bit F are interpreted as follows :

| *M* | *F* | *Mnemonic* | *Meaning* |
|---|---|---|---|
| 0 | 0 | pop %r, {reglist}, #offset | Load from stack memory to selected R6-R14.<br>Copy %r (Rs) to R1. |
| 0 | 1 | pop #imm, {reglist}, #offset | Load from stack memory to selected R6-R14.<br>Copy Imm to R1. |
| 1 | 0 | pop.ret %r, {reglist}, #offset | Load from stack memory to selected R6-R14.<br>Copy %r (Rs) to R1.<br>Copy LR (R14) to PC.<br>Set flags according to new R1. |
| 1 | 1 | pop.ret #imm, {reglist}, #offset | Load from stack memory to selected R6-R14.<br>Copy Imm to R1.<br>Copy LR (R14) to PC.<br>Set flags according to new R1. |

When Rs/Imm is an immediate; 0 represents –1, 1-15 represent 1-15. If the return value wanted is 0, use %r0.

#### 8.10.7.2.1 pop

The operation sequence is :

- Copy return value (register or immediate) to R1
- Load memory into selected registers. %r6 first from the lowest address. %r14 last, from the highest address.

- Increase stack pointer (%r15) by (offset + 4n), where n is the number of selected registers. This action may be distributed through the instruction.
- Set flags based on a comparison of R1 with 0.

The assembler syntax options for pop are :

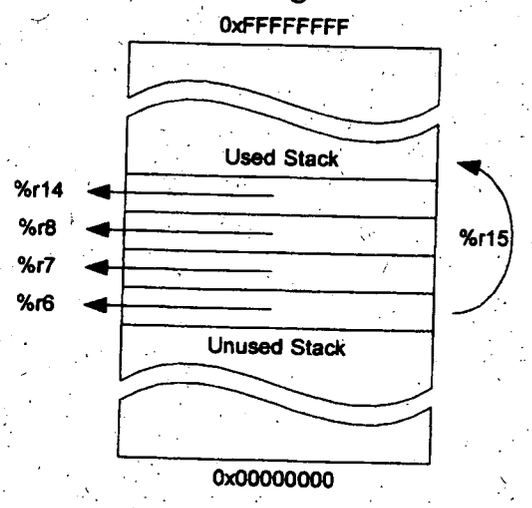
```
pop   #imm, {register_list}, #offset[11:2]u // Return value in R1 = immediate
pop   %r, {register_list}, #offset[11:2]u   // Return value in R1 = register
```

*8.10.7.2.1.1 pop example 1*

```
pop   #7, {%r6, %r7, %r8, %r14}, #0 // Registers must be from low to high
```

This is equivalent to the following sequence of instructions :

```
mov.i  %r1, #7              // Copy 7 to R1 (return register)
ld     %r6, @(0, %r15)      // Load R6 from stack memory
add.i  %r15, %r15, #4       // Increase Stack Pointer for 1 register
ld     %r7, @(0, %r15)      // Load R7 from stack memory
add.i  %r15, %r15, #4       // Increase Stack Pointer for 1 register
ld     %r8, @(0, %r15)      // Load R8 from stack memory
add.i  %r15, %r15, #4       // Increase Stack Pointer for 1 register
ld     %r14, @(0, %r15)     // Load R14 from memory
add.i  %r15, %r15, #4       // Increase Stack Pointer for 1 register
cmp.i  %r1, #0              // Set flags
```



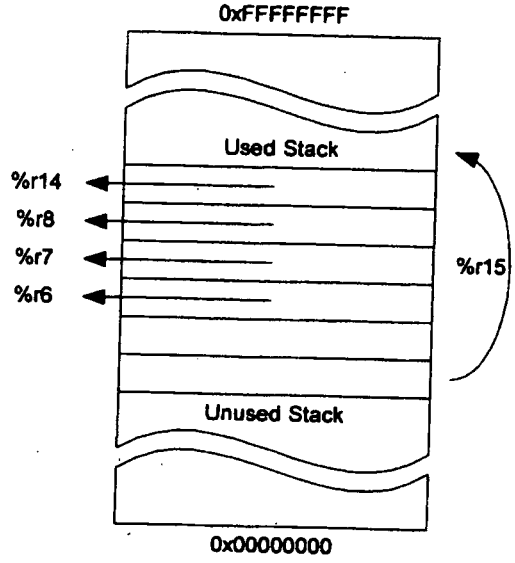

*8.10.7.2.1.2 pop example 2*

```
pop   %r9, {%r6, %r7, %r8, %r14}, #8 // Registers must be from low to high
```

This is equivalent to the following sequence of instructions :

```
mov.r  %r1, %r9             // Copy R9 to R1 (return register)
add.i  %r15, %r15, #8       // Increase Stack Pointer by #offset[11:2]u
```

```
ld      %r6, @(0, %r15)     // Load R6 from stack memory
add.i   %r15, %r15, #4      // Increase Stack Pointer for 1 register
ld      %r7, @(0, %r15)     // Load R7 from stack memory
add.i   %r15, %r15, #4      // Increase Stack Pointer for 1 register
ld      %r8, @(0, %r15)     // Load R8 from stack memory
add.i   %r15, %r15, #4      // Increase Stack Pointer for 1 register
ld      %r14, @(0, %r15)    // Load R14 from memory
add.i   %r15, %r15, #4      // Increase Stack Pointer for 1 register
cmp.i   %r1, #0             // Set flags
```




8.10.7.2.2 pop.ret

- Copy return value (register or immediate) to R1
- Load memory into selected registers. %r6 first from the lowest address. %r14 last, from the highest address.
- Increase stack pointer (%r15) by (offset + 4n), where n is the number of selected registers. This action may be distributed through the instruction.
- Set flags based on a comparison of R1 with 0.
- Copy %r14 (link register) into %pc (program counter). This is done even if %r14 is not in the register list.

The assembler syntax options for pop are :

```
pop.ret #imm, {register_list}, #offset[11:2]u // Return value in R1 = immediate
pop.ret %r, {register_list}, #offset[11:2]u   // Return value in R1 = register
```

*8.10.7.2.2.1 pop.ret example 1*

pop.ret #-1, {%r6, %r7, %r8, %r14}, #0 // Registers must be from low to high

This is equivalent to the following sequence of instructions :

```
mov.i  %r1, #-1          // Copy –1 to R1 (return register)
ld     %r6, @(0, %r15)   // Load R6 from stack memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
ld     %r7, @(0, %r15)   // Load R7 from stack memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
ld     %r8, @(0, %r15)   // Load R8 from stack memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
ld     %r14, @(0, %r15)  // Load R14 from memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
cmp.i  %r1, #0           // Set flags
bra.r  %r14              // Return to address in R14 (Link Register)
                         // and set condition codes for R1.
```

*8.10.7.2.2.2 pop.ret example 2*

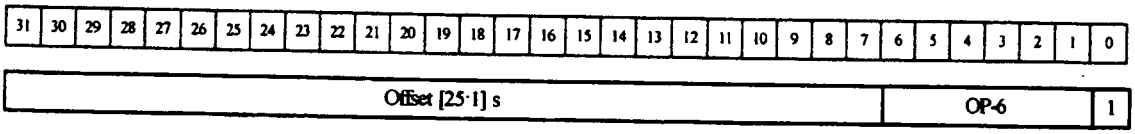
pop.ret %r11, {%r6, %r7, %r8, %r14}, #8 // Registers from low to high

This is equivalent to the following sequence of instructions :

```
mov.r  %r1, %r11         // Copy R9 to R1 (return register)
add.i  %r15, %r15, #8    // Increase Stack Pointer by #offset[11:2]u
ld     %r6, @(0, %r15)   // Load R6 from stack memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
ld     %r7, @(0, %r15)   // Load R7 from stack memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
ld     %r8, @(0, %r15)   // Load R8 from stack memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
ld     %r14, @(0, %r15)  // Load R14 from memory
add.i  %r15, %r15, #4    // Increase Stack Pointer for 1 register
cmp.i  %r1, #0           // Set flags
bra.r  %r14              // Return to address in link register
                         // and set condition codes for R1.
```

## 8.10.8 mov Instructions

### 8.10.8.1 mov.p – PC-relative Offset




This instruction adds a 25-bit signed offset to the current value of the 32-bit Program Counter (PC). The 32-bit result is put into the selected register. It is available for registers R1-R14. There is a separate Opcode for each of these registers.

The offset is interpreted as bits[25:1]. bit[0] = 0. This allows PC relative addresses to be formed on 2byte boundaries (which is the resolution needed for instructions in XAP3). The signed offsets are from –32MB to +32MB.

The resultant address in the register is the absolute 32-bit address. This can be used as Ra in a ld, st, swap, ldm or stm instruction. It could also be used as a pointer or function address.

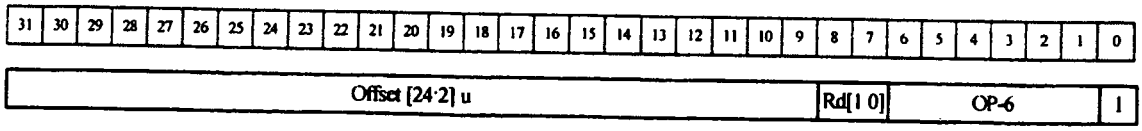
This instruction is very powerful as it can form any of these long addresses in a single cycle. The fact that it is PC-relative makes it suitable for generating Position Independent Code (PIC) or Relocatable Code.

This instruction does not update any flags.

```
mov.p  Rd, #label
mov.p  Rd, #offset[25:1]s      // Use the byte offset (bottom bit must be 0)
       mov.p  %r14, #label36   // Move (PC-relative label36) to R14
       mov.p  %r1, #0xffe132   // Move (PC + 0xfffe132) to R1
```

The offset in the instruction is the address difference between this mov.p instruction and the label. After the mov.p has been executed, Rd will contain the absolute address of the label. This will work, regardless of what address the whole program has been loaded from, because it is PC-relative.

### 8.10.8.2 mov.g – GP-relative Offset




This instruction adds a 23-bit unsigned offset to the current value of the Global Pointer (GP). The 32-bit result is put into the selected register. It is available for registers R1-R14.

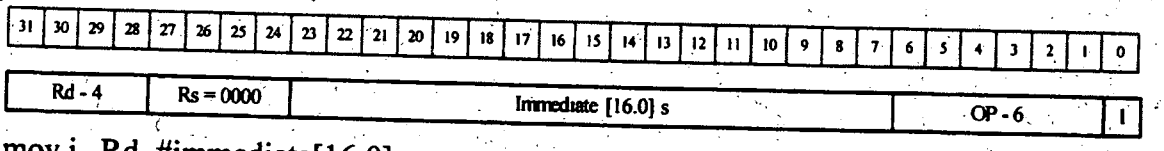
There is a separate Opcode for each group of 4 registers. The least significant 2 bits of OP represent the most significant 2 bits of Rd, ie. OP[1:0] represent Rd[3:2].

mov.g Rd, #label

mov.g %r3, #label22 // R3 = absolute address of GP-relative label22

The offset is interpreted as bits[24:2]. bits[1:0] = 0. This allows GP relative addresses to be formed on 4byte boundaries (which is the resolution needed for data in XAP3). Unsigned offsets are from 0 to 32MB.

### 8.10.8.3 mov.i - Low Immediate




mov.i Rd, #immediate[16:0]s // Sets all 32 bits of Rd

mov.i %r8, #-356 // Load –356 into register R8

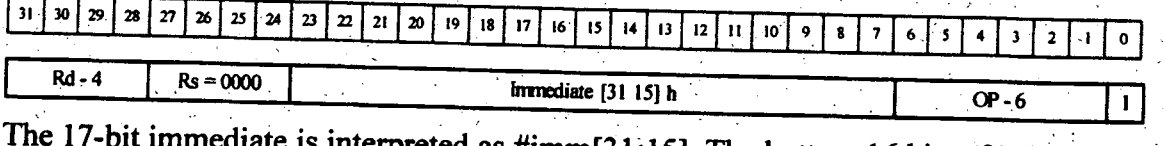
The processor does not explicitly support such an instruction. It will actually be mapped to the following :

or.i %r8, %r0, #-356 // This will set all 32 bits in R8

This means that the disassembler will display the or.i instruction, not the mov.i. This is what will be seen when the program is simulated in xIDE.

In general the compiler will generate mov.r instead of mov.i for #0 because of better code size.

### 8.10.8.4 mov.h - High Immediate




The 17-bit immediate is interpreted as #imm[31:15]. The bottom 16 bits of Rd will all be set to #imm[15]. The top 16 bits of Rd will be set to #imm[31:16]. The immediate in the assembler instruction must therefore set the bottom 16 bits to 0x0000 or 0xffff.

mov.h Rd, #immediate[31:15]s // Sets all 32 bits of Rd

mov.h %r9, #0xab750000 // load Hex ab750000 into register R9

mov.h %r9, #0xab75ffff // load Hex ab75ffff into register R9

In practice it will always be easier to specify the high immediate as a Hex number.

The processor does not explicitly support mov.h. It will actually be mapped to the following :

or.h %r9, %r0, #0xab75ffff // This will set all 32 bits in R8

This means that the disassembler will display the or.h instruction, not the mov.h. This is what will be seen when the program is simulated in xIDE.

If the programmer wants to set all 32 bits in a register with an immediate (e.g 0x12345678), he should use the following sequence of 2 instructions :

```
mov.i  %r8, 0x5678            // sets bottom 16 bits to 0x5678 (and top16 to 0)
or.h   %r8, %r8, 0x12340000 // sets top 16 bits to 0x1234 (leaving bottom 16)
```

### 8.10.8.5 mov.r - Register

mov.r Rd, Rt

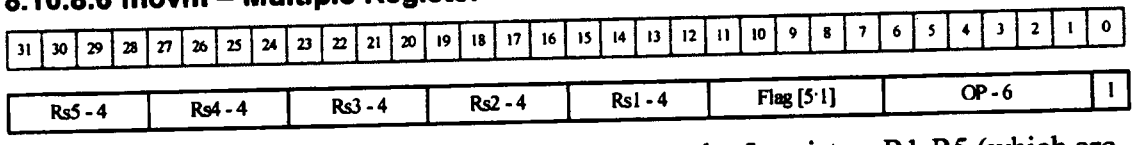
```
mov.r  %r9, %r11              // Copy register R11 to register R9
```

The processor does not explicitly support such an instruction. It will actually be mapped to the following :

```
or.r   %r9, %r0, %r11         // This will actually copy R11 to R9
```

This means that the disassembler will display the or.r instruction, not the mov.r. This is what will be seen when the program is simulated in xIDE.

### 8.10.8.6 movm – Multiple Register




These 2 instructions copy registers or immediates to the 5 registers R1-R5 (which are used to pass C function arguments). Any mixture of immediates or register sources can be used. The immediate values can be (-1, 1, 2 ... 15). If you want to set a register to 0, use %r0.

The instruction formats are :

```
movm (Rs/imm4)1, (Rs/imm4)2, (Rs/imm4)3, (Rs/imm4)4, (Rs/imm4)5
        // Copies to R1 first and R5 last
movm.x (Rs/imm4)1, (Rs/imm4)2, (Rs/imm4)3, (Rs/imm4)4, (Rs/imm4)5
        // Copies to R5 first and R1 last
```

There must always be 5 arguments after the mnemonic. This is necessary as the allocation to the 5 registers R1-R5 is implied by sequence.

If the source (Rs) and destination (Rd) registers are the same, it means 'do not copy anything to Rd'. This mechanism enables movm to copy values to 1, 2, 3, 4 or 5 of R1-R5. In practice, the compiler will not use movm to copy 1 value, as it will use mov.r instead.

Code examples are :

```
movm  %r7, #7, %r10, %r3, %r12    // in this order; copy R7 to R1, 7 to R2,
```

```
                                        // R10 to R3, R3 to R4, R12 to R5.
movm.x  %r7, #7, %r10, %r3, %r12        // in this order; copy R12 to R5, R3 to R4,
                                        // R10 to R3, 7 to R2, R7 to R1.
movm  %r1, %r9, #-1, %r4, %r5           // copy R9 to R2, then –1 to R3.
                                        // nothing is copied to R1, R4 or R5.
```

Flag[5:1] indicate whether Rs[5:1] should be interpreted as Register (0) or Immediate (1). e.g :

```
Flag[5:1] = 0b01101   // Rs4, Rs3 and Rs1 are immediates.
                      // Rs5 and Rs2 are registers.
```

Possible immediate values are :

| | |
|---|---|
| -1 | represented as 0 in the Rs? field |
| 1 to 15 | represented as 1 to 15 in the Rs? field. |

If the user wants to set a return value of 0, he should use %r0.

### 8.10.9 ALU Instructions

The following arithmetic and logic instructions are provided. Each one specifies a destination register (Rd) in which to write back the result, and a source register (Rs) which provides one of the instruction operands. The second input operand may come from another register (Rt), or be specified as a 17-bit immediate value (high or low).

add.*, sub.*, sub.x.*

32-bit addition, subtraction and subtraction with carry. For sub.x the order of the operands is eXchanged (reversed), enabling both (register – immediate for `add.i`) and (immediate – register with `sub.x.i`) to be performed.

- C (carry flag) set if the result of the unsigned operation is not correct
- V (overflow flag) set if the result of the signed operation is not correct

and.*, or.*, xor.*

32-bit logical operations.

mult.i, mult.r, mult.sh.r, mult.16s.*, mult.16u.*, div.s.r, div.u.r, rem.s.r, rem.u.r

| | |
|---|---|
| mult.i, mult.r | 32-bit by 32-bit multiply to give 32-bit result (bottom 32 bits) |
| mult.sh.r | 32-bit by 32-bit multiply then shift right to give 32-bit result |
| mult.16s.* | 16-bit by 16-bit multiply to give 32-bit result<br>Both inputs treated as 16-bit signed variables. |
| mult.16u.* | 16-bit by 16-bit multiply to give 32-bit result<br>Both inputs treated as 16-bit unsigned variables. |
| div.s.r | 32-bit by 32-bit signed divide to give 32-bit result |
| div.u.r | 32-bit by 32-bit unsigned divide to give 32-bit result |
| rem.s.r | 32-bit by 32-bit signed divide to give 32-bit remainder |
| rem.u.r | 32-bit by 32-bit unsigned divide to give 32-bit remainder |

*.u instructions treat all operands (input and output) as unsigned.

#### 8.10.9.1 Rules for Divide and Remainder

Both signed and unsigned divide where the denominator is zero will throw a DivideByZero exception in user mode.

When a divide by zero is attempted in one of the privileged modes, the division does not take place and the destination register remains unchanged. The N and Z flags are updated according to the contents of the destination register. For signed divisions the overflow flag is set and for unsigned divisions the carry flag is set.

div will be executed fortran-style :

- Make both operands unsigned
- Do the division
- Generate the correct sign for the answer

As defined in ISO 9899:1999 (E) C standard; all div and rem (signed and unsigned) work together such that :

- ((a/b)*b) + (a%b) = a

This should be true for (div.s.r with rem.s.r) and for (div.u.r with rem.u.r).

The signed div and rem truncate towards zero :

- div.s.r (5 / 2) = 2, (-5 / 2) = -2, (5 / -2) = -2, (-5 / -2) = 2.
- rem.s.r (5 % 2) = 1, (-5 % 2) = -1, (5 % -2) = 1, (-5 % -2) = -1.
- The special div.s.r case of (0x80000000 / -1) should give the result 0x80000000 and set the overflow flag.
- The div.s.r instruction will set the overflow flag if the result is incorrect.

The unsigned div and rem also truncate towards zero :

- div.u.r (5 / 2) = 2
- rem.u.r (5 % 2) = 1
- The div.u.r instruction will set the carry flag if the result is incorrect.

#### 8.10.9.2 Reg-Reg-Immediate

There are 2 types of immediate instruction; .i low immediates, .h high immediates. In most cases, the immediate is a 17-bit value that is extended to 32-bits before being used as an operand. The exceptions are the following instructions that use unextended 8-bit or 16-bit immediates :

- cmp.8*.i
- cmp.16*.i
- cmp.16*.h
- mult.16*.i

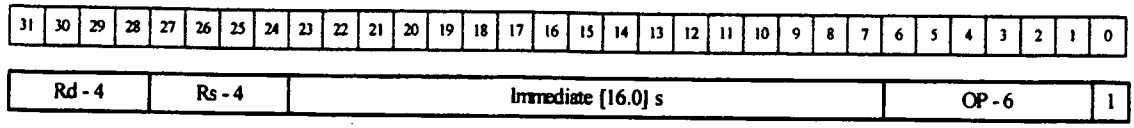
The remaining instructions (with 17-bit immediates extended to 32-bit) should obey the following constraints :

- .i low immediate - top 16 bits must all be 0 or 1.
- .h high immediate - bottom 16 bits must all be 0 or 1

The assembler code should always contain 32-bit immediates that are within range for the associated instruction. The compiler will generate assembler code using XAP3 instructions immediates that are within range for the associated instruction.

If the immediate needs more than 17 bits precision, the compiler will form a 32-bit immediate in a register (using mov.i then or.h) and then use a .r instruction to use the register.

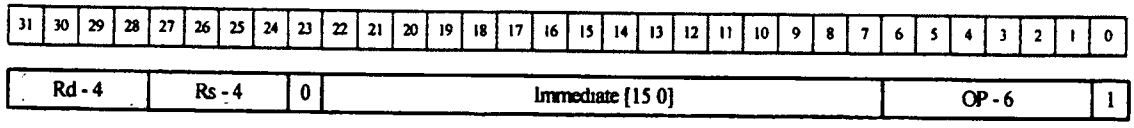
The instructions are formatted as follows for all ALU .i and .h immediate instructions :




For most .i low immediate instructions, data[31:16] = immediate[16] repeated.

add.i Rd, Rs, #immediate[16:0]s

add.i %r5, %r7, #1260 // immediate will be extended to 32-bit




For mult.16s.i and mult.16u.i, the immediate is 16 bits only and not sign-extended.

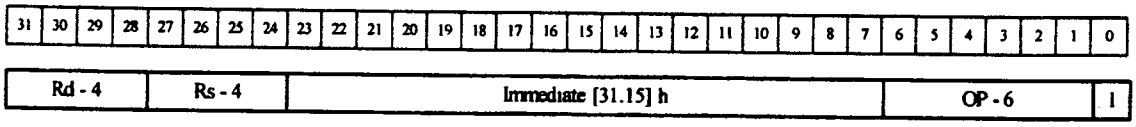
mult.16s.i Rd, Rs, #immediate[15:0] // signed 16-bit multiply

mult.16s.i %r5, %r7, #-1260 // immediate will not be extended

mult.16u.i Rd, Rs, #immediate[15:0] // unsigned 16-bit multiply

mult.16u.i %r5, %r7, #0xf1b3 // immediate will not be extended

The instruction formats for cmp.8*.i, cmp.16*.i and cmp.16*.h are shown in the next section on Compare instructions.

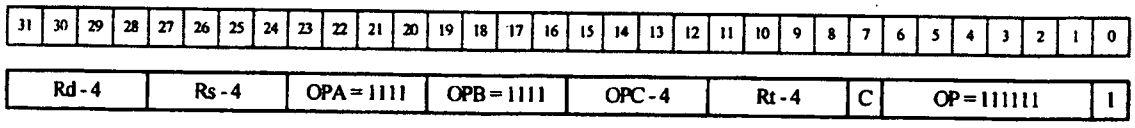



For .h high immediate instructions, data[15:0] = immediate[15] repeated.

add.h Rd, Rs, #immediate[31:15]h

add.h %r3, %r12, #0xAB7DFFFF // immediate will be extended to 32-bit

### 8.10.9.3 Reg-Reg-Reg




add.c.r Rd, Rs, Rt // add.r and sub.r instructions are available in forms
// with and without Carry

xor.r Rd, Rs, Rt // carry is not needed for the rest of the ALU instructions

```
add.r   %r3, %r4, %r0       // equivalent to (mov.r %r4, %r3), though would
                            // not be used for this as no 16-bit form available
mult.r  %r3, %r4, %r12      // R3 = R4 * R12
xor.r   %r3, %r4, %r6       // R3 = R4 ^ R6
and.r   %r5, %r4, %r7       // R5 = R4 & R7
```

### 8.10.10 Compare Instructions

The comparison instructions are used to set the processor's Flags. These are used by the conditional branch instructions. There are 32, 16 and 8-bit compare instructions.

The 32-bit compare instructions (cmp.*.i, cmp.*.h, cmp.*.r) operate on all 32-bits

The 16-bit compare instructions (cmp.16*.i, cmp.16*.r) operate on the bottom 16-bits only.

The 16-bit compare high instructions (cmp.16*.h) operate on the top 16-bits only.

The 8-bit compare instructions (cmp.8*.i, cmp.8*.r) operate on the bottom 8-bits only.

Note that the order in which the two arguments are used in the comparison can be specified so that together the branch and comparison instructions allow all conditions to be tested for.

#### 8.10.10.1 Compare Assembler Format

There are 2 forms of the cmp instruction (immediate, register). These take parameters for OperandSize, CarryBit and Operand Order. These are interpreted as follows :

| *Operand Size* | *Assembler Parameter* | *Meaning* |
|---|---|---|
| 00 | 8 | 8-bit |
| 01 | 16 | 16-bit |
| 10 | 32 | 32-bit |
| 11 | - | Reserved |

| *Carry Bit* | *Assembler Parameter* | *Meaning* |
|---|---|---|
| 0 | nocarry | Without Carry |
| 1 | carry | With Carry |

| *Operand Order* | *Assembler Parameter* | *Flags set on according to ...* |
|---|---|---|
| 0 | normal | Rs – immediate[16:0]s |
| 1 | eXchange | immediate[16:0]s – Rs |

OperandOrder is only used for cmp.*.i and cmp.*.h immediate compare instructions. It is meaningles and therefore not used for the cmp.*.r register compare instructions.

Assembler mnemonics for the cmp instruction are of the form :

- cmp.*Parameters.Type*

The default parameters for the cmp instruction are :

- 32 bit
- No Carry

- Normal Order (not reversed)

Non-default parameters available are :

- 16 16-bit
- 8 8-bit
- c With Carry
- x eXchange (Reverse) Order

Types specify the type of second ALU input. Types available (like all the major instructions) are :

- i Low Immediate
- h High Immediate
- r Register

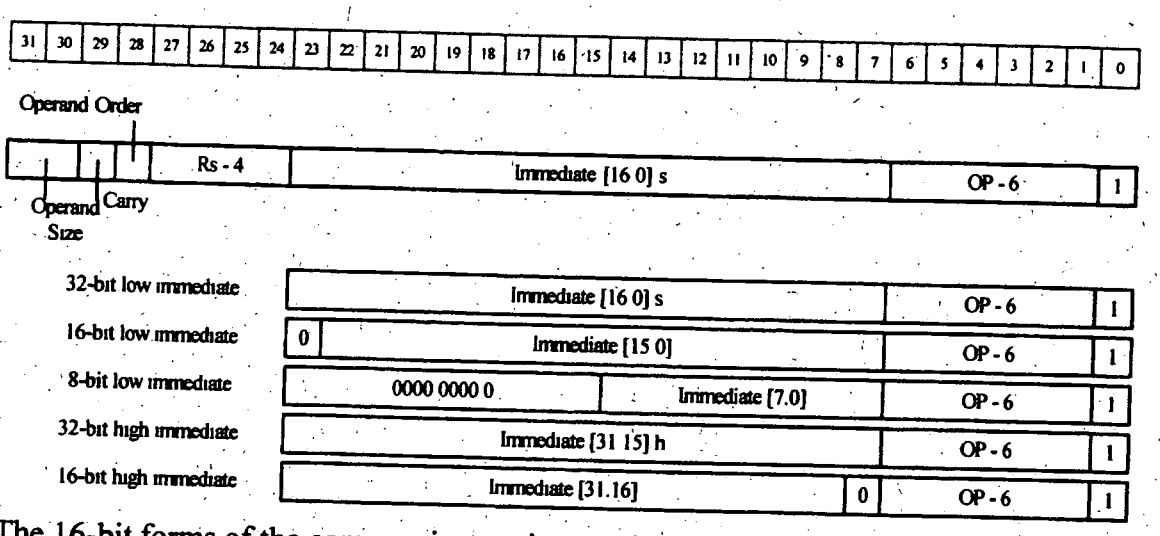
So example cmp instructions are :

- cmp.r Register, 32-bit, No Carry, Normal Order
- cmp.c.i Low Immediate, 32-bit, With Carry, Normal Order
- cmp.8xc.i Low Immediate, 8-bit, With Carry, eXchange Order

#### 8.10.10.2 Compare Immediate

There are 5 formats for the compare immediate instructions :

| *Format* | *Instructions* | *Compare Rs with ...* | *Extend Immediate ?* |
|---|---|---|---|
| 32-bit low immediate | cmp.i, cmp.c.i, cmp.x.i, cmp.xc.i | imm[31:0] | imm[16:0] is extended upwards to 32-bit. |
| 16-bit low immediate | cmp.16.i, cmp.16c.i, cmp.16x.i, cmp.16xc.i | imm[15:0] | No |
| 8-bit low immediate | cmp.8.i, cmp.8c.i, cmp.8x.i, cmp.8xc.i | imm[7:0] | No |
| 32-bit high immediate | cmp.h, cmp.c.h, cmp.x.h, cmp.xc.h | imm[31:0] | imm[31:15] is extended downwards to 32-bit. |
| 16-bit high immediate | cmp.16.h, cmp.16c.h, cmp.16x.h, cmp.16xc.h | imm[31:16] | No |




The 16-bit forms of the compare instruction can be used to implement a comparison with a 32-bit immediate. The lower 16-bits of the immediate are first compared with the register. Then the upper 32-bits are compared with the register and the carry flag. For example compare R4 with 0x12345678:

```
cmp.16.i    %r1, #0x5678
cmp.16c.h   %r1, #0x12340000
```

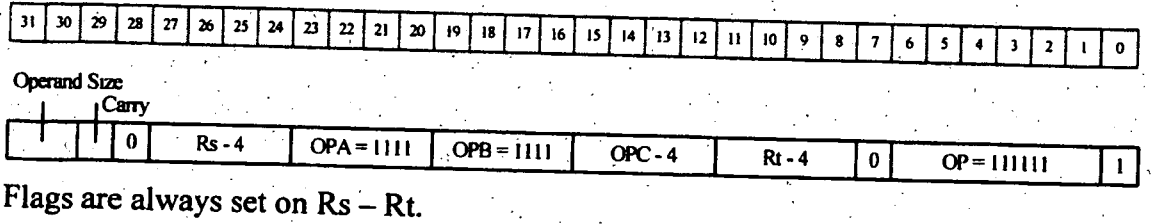
The cmp.16*.h instructions compares bits [31:16] of the register with bits [31:16] of the immediate.

For cmp.16c.h, the Z flag is only set if (the result of the subtraction was 0) and (the Z flag was already 1).

In general the compiler will generate cmp.r instead of cmp.i for #0 because this compacts better.

```
cmp.*.i     Rs, #immediate[16:0]s
    cmp.c.i     %r3, #0xAbc3 // 32-bit compare, With Carry
cmp.*.h     Rs, #immediate[31:15]h
    cmp.16x.h   %r3, #-35    // 16-bit compare high, eXchange order, No Carry
```

### 8.10.10.3 Compare Register




Flags are always set on Rs – Rt.

16-bit register compares only compare the bottom 16 bits of Rs and Rt.

8-bit register compares only compare the bottom 8 bits of Rs and Rt.

cmp.*.r Rs, Rt

```
cmp.c.r     %r3, %r7    // 32-bit compare with Carry of (R3 – R7)
cmp.16.r    %r7, %r3    // 16-bit compare without Carry of (R7 – R3)
```

### 8.10.11 Shift and Rotate Instructions

shiftl.r, shiftr.s.r, shiftr.u.r, rotatel.r

The XAP3 has signed (arithmetic) and unsigned (logical) left and right shift instructions.

The shift instructions use the bottom 5 bits of Rt or the immediate argument as an unsigned shift value. This supports shifts of up to 31 bits to the left or right.

For left shifts the last bit shifted out of the source register will be copied to the carry flag. If the register is shifted by 0, the Carry Flag will not be modified.

For right shifts, the C flag equals the bit last shifted out of bit 0 of Rs.

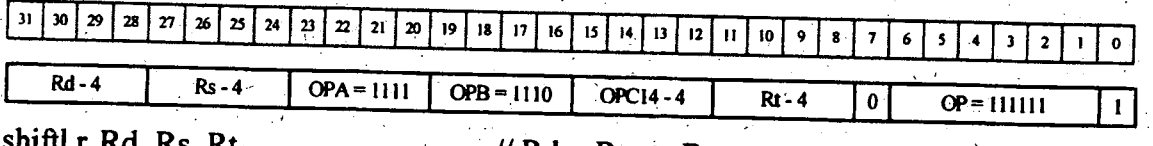
The rotatel instruction uses the bottom 5 bits of Rt or the immediate argument as an unsigned rotate left value. The rotatel instruction does not circulate through the carry bit. It sets the flags as follows :

- C = final value of bit 0 of Rd.
- N = final value of bit 31 of Rd.
- Z = 1 if the contents of Rd = 0, otherwise Z = 0.

#### 8.10.11.1 Reg-Reg-Immediate

These instructions are implemented by the rlinm instruction (see later).

#### 8.10.11.2 Reg-Reg-Reg




```
shiftl.r Rd, Rs, Rt                 // Rd = Rs << Rt
        shiftl.r   %r3, %r2, %r13   // Shift R2 to left by value in R13, store result in R3

        shiftr.s.r %r3, %r2, %r13   // Sign Shift R2 to right by R13, store result in R3
        shiftr.u.r %r3, %r2, %r13   // Unsign Shift R2 to right by R13, store result in R3
        rotatel.r  %r3, %r2, %r13   // Rotate R2 left by value in R13, store result in R3
```

#### 8.10.11.3 rlinm Instruction

Rotate Left Immediate, then aNd with Mask.

There are sign and zero extending versions of this instruction.

This is a very powerful instruction for shifting words and for bit operations. It is used for all the shift immediate instructions. It takes 1 parameter :

- signext 1-bit // 1 = sign extend, 0 = zero extend

and 3 unsigned immediates :

- shift 5-bit // rotate left by 0 to 31 bits
- highbit 5-bit // highbit = 0 to 31
- lowbit 5-bit // lowbit = 0 to 31

Rs is first rotated to the left by #shift.

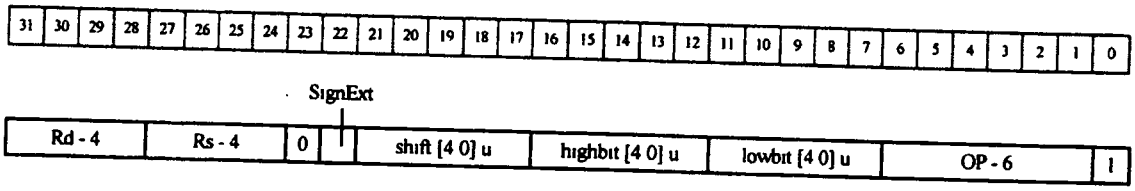
This result is then ANDed with a 32-bit word that consists of :

- Contiguous string of 1s between (and including) #highbit and #lowbit
- 0s for all other bits

This result is then optionally sign or zero extended from #highbit to bit-31 before being stored in Rd.

#highbit must have a value that is greater than or equal to #lowbit.

The C flag is set to bit 0 of Rs after the rotation but before the masking operation.




rlinm.s Rd, Rs, #shift5u, #highbit5u, #lowbit5u

rlinm.z Rd, Rs, #shift5u, #highbit5u, #lowbit5u

rlinm.s %r3, %r7, #4, #23, #15

rlinm.z %r8, %r9, #15, #12, #12

Immediate shifts and rotates are generated as specific forms of the rlinm instruction. The immediate values for the rlinm instruction are computed according to the following table.

| *Pseudo-Instruction* | *rlinm.[sz]* | *Rd, Rs, #shift5u, #highbit5u, #lowbit5u* |
|---|---|---|
| Left shift | rlinm.z | Rd, Rs, #Imm, #31, #Imm |
| Signed right shift | rlinm.s | Rd, Rs, #(32-Imm) & 31, #(31-Imm), #0 |
| Unsigned right shift | rlinm.z | Rd, Rs, #(32-Imm) & 31, #(31-Imm), #0 |
| Rotate left | rlinm.z | Rd, Rs, #Imm, #31, #0 |

### 8.10.12 Special Register Instructions

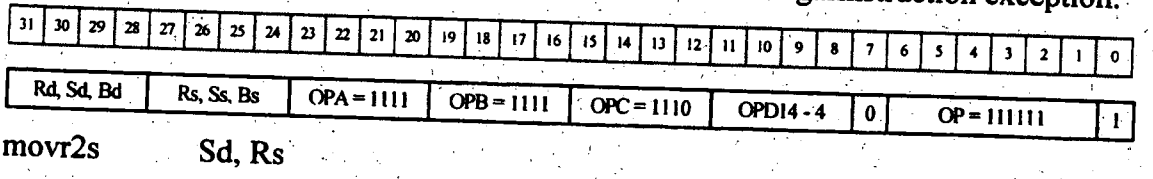
The movr2s and movs2r instructions allow the privileged modes to access special registers that are not part of the normal register model. This includes the flags.

Similarly, the movb2r and movr2b instructions allow the privileged modes to access the breakpoint registers.

| *Instruction* | *Operation* |
|---|---|
| movr2s | Move Normal Register to Special Register |
| movs2r | Move Special Register to Normal Register |
| movr2b | Move Normal Register to Breakpoint Register |
| movb2r | Move Breakpoint Register to Normal Register |

The coding and assembler syntax for the Special Register is shown in section 5.6 and the for the breakpoint registers in section 5.5.

Executing these instructions in User Mode will cause an IllegalInstruction exception.




movr2s Sd, Rs

movs2r Rd, Ss

movr2b Bd, Rs

movb2r Rd, Bs

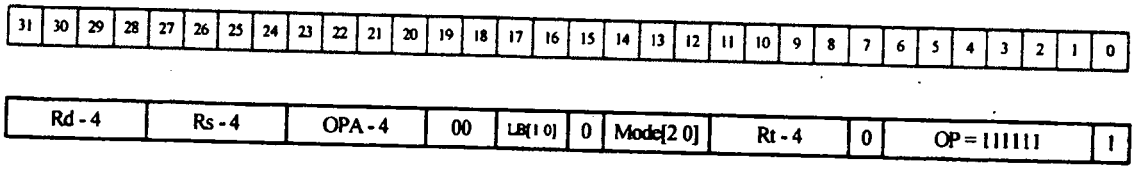
```
movs2r   %r1, %flags      // Move FLAGS to R1
movr2s   %sflags_s, %r2   // Move R2 to SFLAGS_S
movb2r   %r1, %brk2       // Move BRK2 to R1
movr2b   %brk1, %r2       // Move R2 to BRK1
```

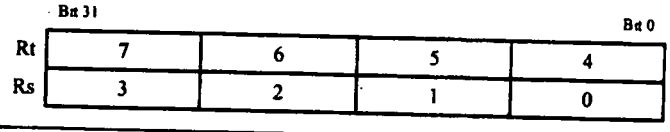
## 8.10.13 DSP Instructions

### 8.10.13.1 flip.*.r Instruction

flip.*.r Rd, Rs, Rt, #lowbyte (Rd = 4 contiguous bytes from middle of Rt:Rs)

The instruction first selects 4 contiguous bytes from 8. Then it optionally swaps bits or bytes for endian conversions (little to big and vice-versa).




| *LowByte* | *Bytes copied to Rd* |
|---|---|
| 0 | 3210 |
| 1 | 4321 |
| 2 | 5432 |
| 3 | 6543 |

Having selected the 32-bit word, it can then be bit or byte swapped before being copied into Rd :

| *Mode[2:0]* | *Mnemonic* | *Meaning* | *Swap Details* |
|---|---|---|---|
| 0 | flip.r | No swaps | None |
| 1 | flip.x.r | 32-bit bitswap | bits 0/31, 1/30 etc |
| 2 | flip.16x.r | 16-bit bitswap | bits 0/15, 1/14; 16/31, 17/30 etc |
| 3 | flip.8x.r | 8-bit bitswap | bits 0/7, 1/6; 8/15; 16/23; 24/31, 25/30 etc |
| 4 | flip.b.r | 32-bit byteswap | bytes 0/3 and 1/2 |
| 5 | flip.16b.r | 16-bit byteswap | bytes 0/1 and 2/3 |
| 6 | | | |
| 7 | | | |

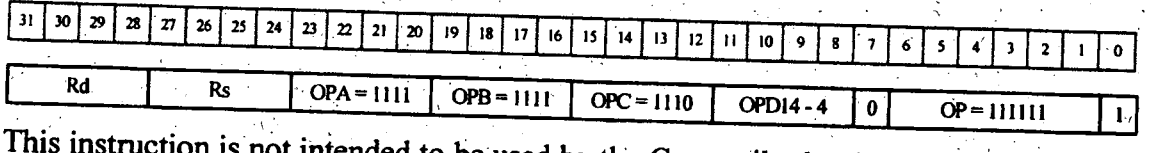
This instruction will not be used by the C compiler. However it is a very powerful assembler instruction for performing a variety of byte-selection then byte-swapping or bit-swapping functions in a single cycle instruction. It will be very useful for

communications stacks and for porting between systems that use different bit or byte endian conventions (little or big). The XAP3 always uses little-endian byte-ordering.

flip.r Rd, Rs, Rt, #lowbyte2 (Rd = 4 contiguous bytes from middle of Rt:Rs)

```
flip.r   %r2, %r7, %r14, #2   // copy bytes 5432 from {R14, R7} to R2
flip.r   %r6, %r7, %r8, #0x3  // copy bytes 6543 from {R8, R7} to R6
flip.x.r %r6, %r7, %r8, #0x3  // select bytes 6543 from {R8, R7}
                              // then do a 32-bit bitswap and copy to R6
flip.b.r %r6, %r7, %r8, #0x3 // select bytes 6543 from {R8, R7}
                              // then do a 32-bit byteswap and copy to R6
flip.16b.r %r6, %r7, %r8, #0x3 // select bytes 6543 from {R8, R7}
                              // then do a 16-bit byteswap and copy to R6
```

### 8.10.13.2 abs.r



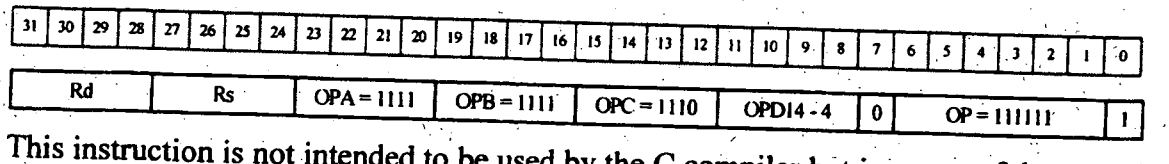

This instruction is not intended to be used by the C compiler but is a useful assembler instruction. It operates on a 32-bit register :

```
abs.r    Rd, Rs    // Rd = absolute (Rs)
```

Rs ≥ 0 : Rd = Rs

Rs < 0 : Rd = -Rs

### 8.10.13.3 msbit.r




This instruction is not intended to be used by the C compiler but is a powerful assembler instruction that is useful for DSP. It operates on a 32-bit register :

```
msbit    Rd, Rs
```

Rd is set with the position of the highest 1 in the word. It should be used on positive or unsigned numbers. The value returned in Rd is (1 + highest bit to contain a 1 in Rs). This is the minimum number of bits that must be shifted to the right to make Rs = 0.

| *Rs* | *Rd* |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 4 | 3 |
| 7 | 3 |
| 8 | 4 |
| 0x400 | 11 |
| 0x100000 | 21 |
| 0x3FFFFFFF | 30 |
| 0x40000000 | 31 |
| 0x80000000 | 32 |
| 0xFFFFFFFF | 32 |

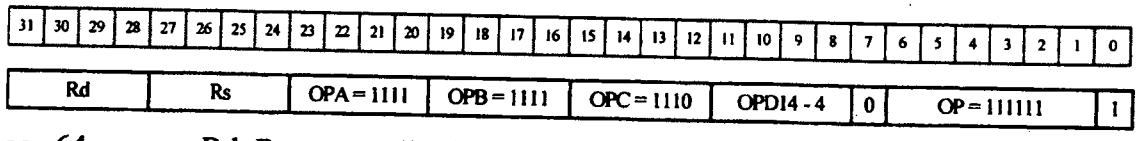
#### 8.10.13.4 acc.64.r

There are 2 64-bit accumulators :

A1 = {R11, R10} // Referenced in instructions as %r10

A0 = {R9, R8} // Referenced in instructions as %r8

These are useful for I (in phase) and Q (quadrature) filters.



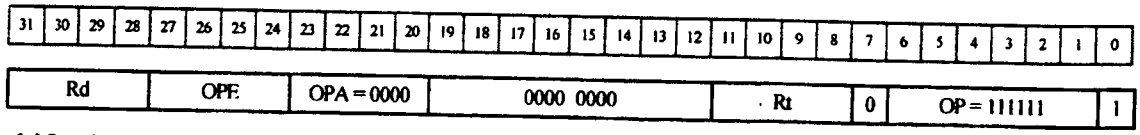

acc.64.r Rd, Rs // Rd += Rs

// Rd = 64-bit = R8 or R10, Rs = 32-bit = any register

#### 8.10.13.5 64-bit shift instructions

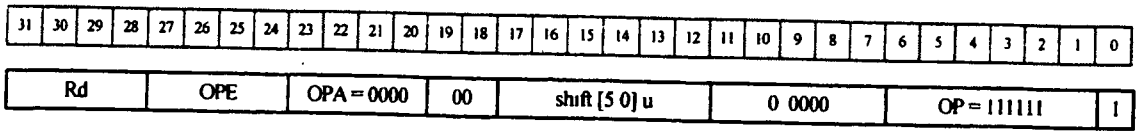
For all the 64-bit shift instructions :

- Rd = 64-bit = R8 or R10, Rt = bottom 6-bits of any register




shiftr.64s.r Rd, Rt // Rd >>= Rt

shiftl.64.r Rd, Rt // Rd <<= Rt

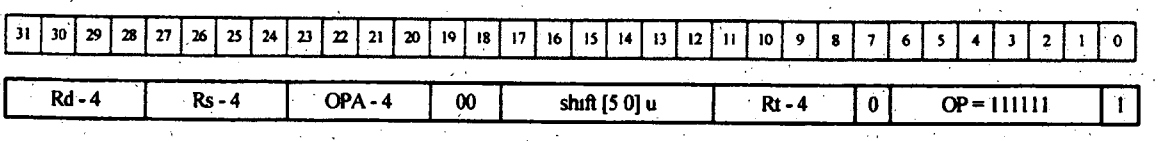


shiftr.64s.i Rd, #shift // Rd >>= shift

shiftl.64.i Rd, #shift // Rd <<= shift

Rd can be shifted left or right by (0 to 63) bits in 1-bit steps. The 6-bit shift amount can be specified in a register (*.r instructions) or in an immediate (*.i instructions). In both cases the number is interpreted as unsigned.

### 8.10.13.6 mult.sh.r




mult.sh.r Rd, Rs, Rt, #shift // Rd = (Rs * Rt) >> shift

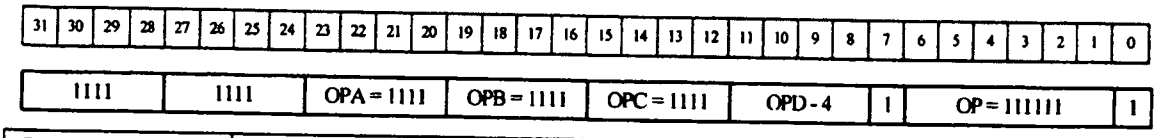
mult.sh.r does the following :

- 32 * 32 signed multiply of Rs and Rt into a temporary 64-bit register.
- Signed shift right the 64-bit register by the #shift parameter (0 to 63).
- Writes the resulting bottom 32 bits to Rd

This is useful when multiplying coefficients and data and normalising to pick out the resultant 32 bits of interest. This can then be added to a 64-bit accumulator with an acc.64.r instruction.

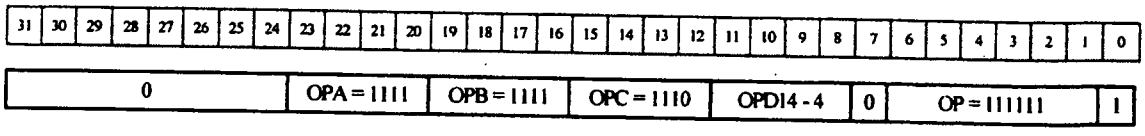
### 8.10.14 System Instructions

The instructions below are part of the OPD encoding.



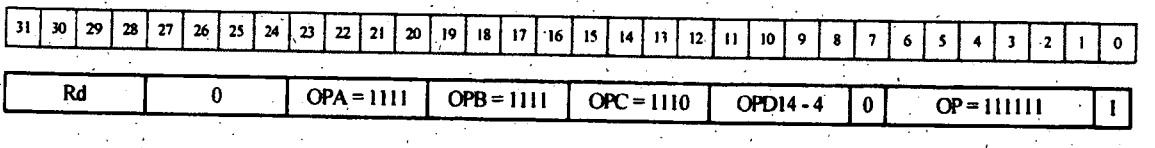

| *Instruction* | *Meaning* |
|---|---|
| rtie | Return from interrupt and exception. The current mode's link register is written to the PC and the SFLAGS register is written to the FLAGS register. This instruction is not permitted in User Mode. |
| brk | If the processor is in User Mode and the B bit of flags is set, the break exception is thrown. Otherwise, if the processor is in debug mode the processor is stopped. |
| halt | Stops the processor always. This instruction is not permitted in User Mode. |
| sleepnop | Puts the processor into sleep mode. The WAKE_UP signal restarts the processor. During the sleep mode no SIF accesses to memory or registers are possible. This instruction is not permitted in User Mode. |
| sleepsif | Puts the processor into sleep mode. The WAKE_UP signal restarts the processor. SIF accesses are possible during the sleep mode. This instruction is not permitted in User Mode. |
| nop | No operation. |
| sif | NOP instruction left clear for SIF read or SIF write. This instruction is not permitted in User Mode. The 32-bit coding of the sif instruction is all 1's, 0xFFFFFFFF. The 16-bit coding of the SIF instruction is all 0's, 0x0000. |

The instructions below are part of the OPD14 encoding.



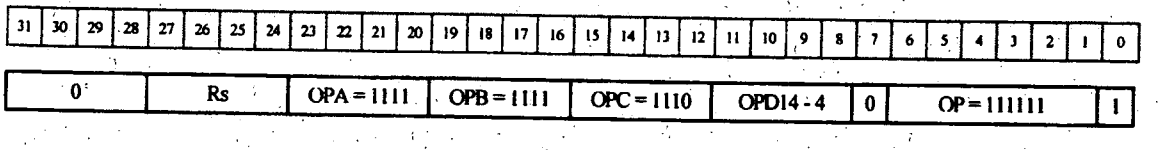

| *Instruction* | *Meaning* |
|---|---|
| irqe | Interrupt Enable. Sets the E bit of the FLAGS register to 1. This instruction is not permitted in User Mode. |
| irqd | Interrupt Disable. Clears the E bit of the FLAGS register to 0. This instruction is not permitted in User Mode. |

The instruction below is part of the OPD14 encoding.



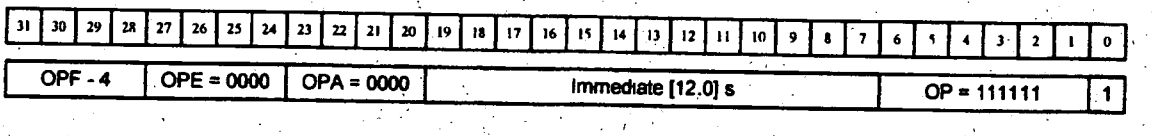

| *Instruction* | *Meaning* |
|---|---|
| irqs Rd | Interrupt Status. |

The instructions below are part of the OPD14 encoding.




| *Instruction* | *Meaning* |
|---|---|
| print.r Rs | xIDE simulator prints value in Rs (interpreted as a char). Hardware treats instruction as NOP. |
| trap.r Rs | Cause the trap exception. Allows User Mode to call Supervisor Mode functions. This is executed as a nop in Supervisor and Interrupt Modes. |

The instruction below is part of the OPF coding:




| *Instruction* | *Meaning* |
|---|---|
| trap.i #imm | Cause the trap exception. Allows User Mode to call Supervisor Mode functions. This is executed as a nop in Supervisor and Interrupt Modes. |

The trap instructions pass a number (in a register or as an immediate), that is intended to be used by an operating system to indicate a particular service or function. The processor copies this number to R1_S. Additional arguments for the function can be passed in the User mode registers R2-R13.

## 8.11 16-bit Instruction Formats

The 16-bit instructions are a subset of the 32-bit instruction set.

They use the same assembler mnemonics but have shorter arguments.

In order to compress the instruction size, many of the 16-bit opcodes assume that Rd is the same as Rs.

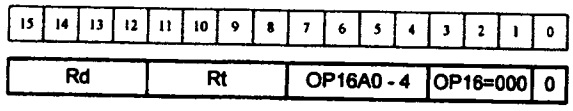
The assembler syntax is the same for 32-bit and 16-bit instructions. The C compiler does not know about which instructions will be 32-bit and which will be 16-bit. The binutils assembler will choose whether to use 32-bit or 16-bit opcodes for particular assembler instructions. It will choose 16-bit instructions whenever possible.

When shorter fields (2 or 3 bit) are used for the registers, a decode table is provided. Register decodes are standard for 4 bit fields (0 = R0, ..., 15 = R15).

### *8.11.1 OP16A0 Register-Register Instructions*

OP16B0, add.r, sub.r, add.i, and.r, or.r, xor.r, mult.r, ld.r, ld.16z.r, ld.8z.r, mov.r, st.r, st.16.r, st.8.r, cmp.r

The 6 Reg-Reg instructions {add.r, sub.r, and.r, or.r, xor.r, mult.r} are in the format :

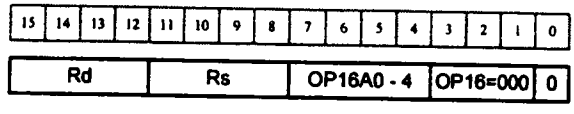



For example, the add.r instruction will be converted to the 32-bit instruction:

- add.r Rd, Rd, Rt

Note that for these 6 instructions, neither Rd nor Rt may be R0. These opcodes will not be compressed to 16-bit instructions and are reserved for future use.

The Reg-Imm instruction add.i instruction has the format:

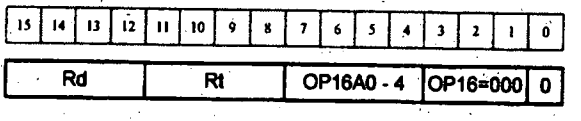



This will be converted to the 32-bit instruction:

- add.i Rd, Rs, #1

Note that the add.i instruction shall not have R0 for Rd or Rs. These opcodes will not be compressed to 16-bit instructions and are reserved for future use.

The mov.r instruction has the format:

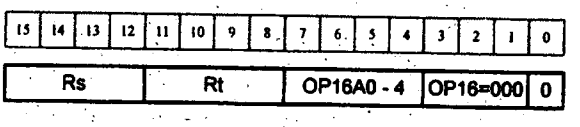



This will be converted to the 32-bit instruction:

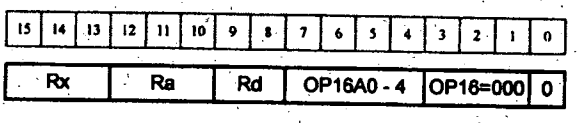
- or.r Rd, %r0, Rt

Note that the or.r instruction shall not have R0 for Rd. These opcodes will not be compressed to 16-bit instructions and are reserved for future use.

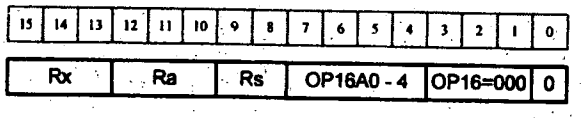
The cmp.r instruction is in the format:




The {ld.r, ld.16z.r, ld.8z.r} instructions are in the format:




The {st.r, st.16.r, st.8.r} instructions are in the format:




The load and store instructions decode the 3 register fields as follows:

Rd /Rs select one of R1-R4:

| *Rd/Rs* | *Meaning* |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

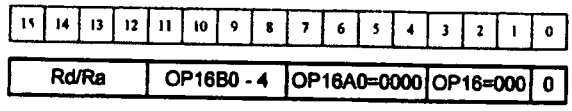
Rx and Ra select one of {R1-R4, R6-R9}:

| *Rx/Ra* | *Meaning* |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |

| | |
|---|---|
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

### 8.11.2 OP16B0 Instructions

sif, nop, brk, xor.i, sub.x.i, bra.r, bsr.r, seq.i, sne.i




The xor.i 16-bit encoding implements:

xor.i Rd, Rd, #0xFFFFFFFF

The sub.x.i 16-bit encoding implements:

sub.x.i Rd, Rd, #0

Both forms of the sne.i and seq.i instructions have similar coding implementations, the only difference being in the value of the immediate:

seq.i Rd, %r0, #1

sne.i Rd, %r0, #1

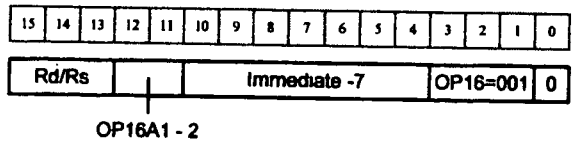
and

seq.i Rd, %r0, #-1

sne.i Rd, %r0, #-1

### 8.11.3 OP16A1 Register-Immediate Instructions

add.i, add.i (SP), cmp.i, mov.i




The 3-bit Rd/Rs field is interpreted as per the table below:

| *Rd/Rs* | *add.i* | *add.i (SP)* | *cmp.i* | *mov.i* |
|---|---|---|---|---|
| 000 | R8 | R15 | R8 | R8 |
| 001 | R1 | R1 | R1 | R1 |
| 010 | R2 | R2 | R2 | R2 |
| 011 | R3 | R3 | R3 | R3 |
| 100 | R4 | R4 | R4 | R4 |

| *Rd/Rs* | *add.i* | *add.i (SP)* | *cmp.i* | *mov.i* |
|---|---|---|---|---|
| 101 | R9 | R5 | R9 | R9 |
| 110 | R6 | R6 | R6 | R6 |
| 111 | R7 | R7 | R7 | R7 |

For the add.i instruction, the immediate is interpreted as immediate[6:0]s with zero meaning +64. Hence, immediate values in the range -64, -63, ... -1, 1, 2, ... 63, 64 can be expressed. This is converted to the 32-bit instruction:

- add.i Rd, Rd, #imm[6:0]s

For the add.i (SP) instruction, the immediate is interpreted as immediate[8:2]s, with zero meaning +256. Hence, immediate values in the range –256, -252, ... -4, 4, 252, 256 can be expressed. This is converted to the 32-bit instruction:

- add.i Rd, %r15, #imm[8:2]s

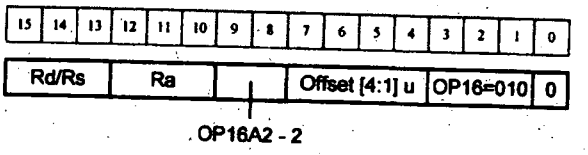
For the cmp.i and mov.i instructions, the immediate is interpreted as immediate[6:0]u with zero meaning -1. Hence, immediate values in the range –1, 1, ..., 127 can be expressed. These are converted to the 32-bit instructions:

- cmp.i Rs, #imm[6:0]u
- or.i Rd, %r0, #imm[6:0]u // equivalent to mov.i

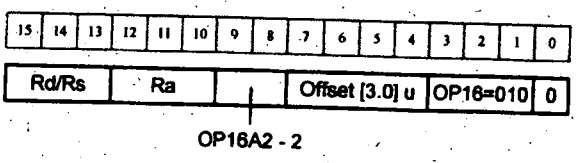
### *8.11.4 OP16A2 16 and 8 bit ld and st Instructions for 16 and 8 bit Data*

ld.8z.i, ld.16z.i, st.8.i, st.16.i

For 16-bit loads and stores the instruction is coded as below:




For 8-bit loads and stores the instruction is coded as below:




Ra selects from the set registers as below.

| *Ra?* | *Meaning* |
|---|---|
| 000 | R15 |
| 001 | R1 |

| | |
|---|---|
| 010 | R2 |
| 011 | R3 |
| 100 | R8 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

For st.8.i and st.16.i, the Rs field is interpreted as below:

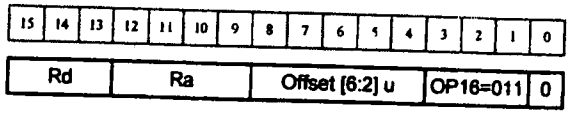
| *Rs?* | *Meaning* |
|---|---|
| 000 | R0 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

For ld.8z.i and ld.16z.i, the Rd field is interpreted as below:

| *Rd?* | *Meaning* |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

### *8.11.5 OP16A3 32-bit ld.i Instructions*

Rd = @(offset7u, Ra)



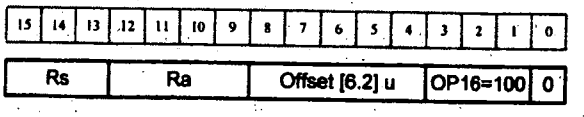

Where Rd selects one of R1-R8 as described below:

| *Rd?* | *Meaning* |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |

| *Rd?* | *Meaning* |
|---|---|
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

### 8.11.6 OP16A4 32-bit st.i Instructions

@(offset7u, Ra) = Rs




Where Rs selects one of R0-R7 as described below:

| *Rs?* | *Meaning* |
|---|---|
| 000 | R0 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

### 8.11.7 OP16A5 movm and bra.p Instructions

#### 8.11.7.1 movm

The movm instruction has the following form:

movm (Rs/imm4)1, (Rs/imm4)2, (Rs/imm4)3, (Rs/imm4)4, (Rs/imm4)5

// Argument 1 = Reg or Imm4 copied to R1

// ...

// Argument 5 = Reg or Imm4 copied to R5

// Copies to R1 first and R5 last

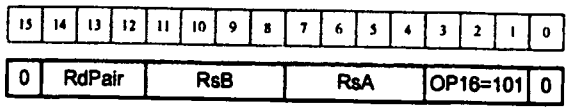
The 32-bit movm instruction can copy to all 5 registers, R1-R5. i.e 5 Rd.

The 16-bit movm instruction can copy to 2 of the registers, R1-R5. Various RdPair codings allow different pairs of Rd to be selected.

The 16-bit movm copies :

- RsA to RdA.

- RsB to RdB




RsA and RsB fields are interpreted as follows :

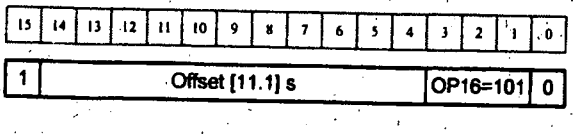
| *Rs?* | *Meaning* |
|---|---|
| 0000 | R0 |
| 0001 | R1 |
| 0010 | R2 |
| 0011 | R3 |
| 0100 | R4 |
| 0101 | R5 |
| 0110 | R6 |
| 0111 | R7 |
| 1000 | R8 |
| 1001 | R9 |
| 1010 | R10 |
| 1011 | R15 |
| 1100 | #4 |
| 1101 | #1 |
| 1110 | #2 |
| 1111 | #3 |

The RdPair field is interpreted as follows :

| *RdPair* | *RdB* | *RdA* |
|---|---|---|
| 000 | R2 | R1 |
| 001 | R3 | R1 |
| 010 | R4 | R1 |
| 011 | R5 | R1 |
| 100 | R3 | R2 |
| 101 | R4 | R2 |
| 110 | R4 | R3 |
| 111 | R5 | R4 |

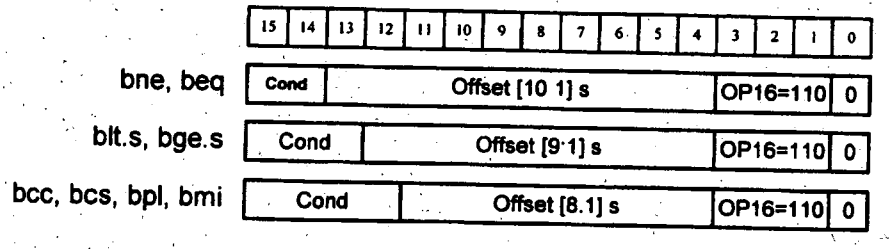
### 8.11.7.2 bra.p

The bra.p instruction has the following form:




### 8.11.8 OP16A6 Conditional Branch Instructions

beq, bne, blt.s, bge.s, bcs, bcc, bmi, bpl

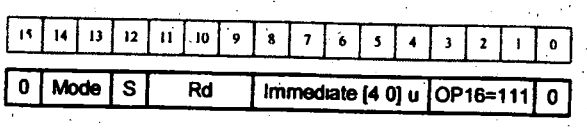



| *Cond* | *Condition* |
|---|---|
| 00XX | beq |
| 01XX | bne |
| 100X | blt.s |
| 101X | bge.s |
| 1100 | bmi |
| 1101 | bpl |
| 1110 | bcs |
| 1111 | bcc |

### 8.11.9 OP16A7 Instructions

#### 8.11.9.1 OP16A7 rlinm Instruction

rlinm.s, rlinm.z




The rlinm instruction relies on two fields for choosing various operating modes:

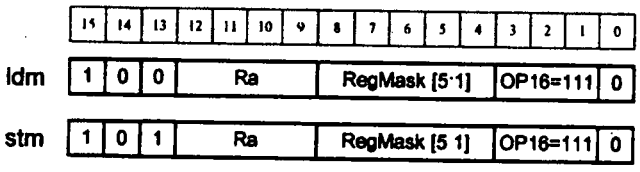
| *Mode* | *Sign* | *32-bit form* | *Equivalent Instruction* |
|---|---|---|---|
| 00 | 0 | #Imm, #(Imm-1), #0 | shiftr.u.i Rd, Rd, #(32-Imm) & 31 |
| 00 | 1 | #Imm, #(Imm-1), #0 | shiftr.s.i Rd, Rd, #(32-Imm) & 31 |
| 01 | 0 | #Imm, #31, #Imm | shiftl.i Rd, Rd, #Imm |

| *Mode* | *Sign* | *32-bit form* | *Equivalent Instruction* |
|---|---|---|---|
| 01 | 1 | do not use – reserved for future use | |
| 10 | 0 | #0, #Imm, #0 | and.i Rd, Rd, #($2^{Imm+1}$-1) |
| 10 | 1 | #0, #Imm, #0 | sext.i Rd, #Imm |
| 11 | 0 | #0, #Imm, #Imm | and.i Rd, Rd, #$2^{Imm}$ |
| 11 | 1 | do not use – reserved for future use | |

Rd selects from the set R1-R8.

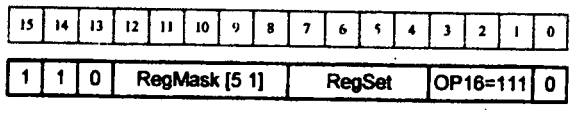
| *Rd?* | *Meaning* |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

### 8.11.9.2 OP16B7 ldm and stm Instructions




The RegMask[5:1] fields match bits RegisterMask[5:1] in the 32-bit form.

### 8.11.9.3 OP16B7 push/push.r Instruction

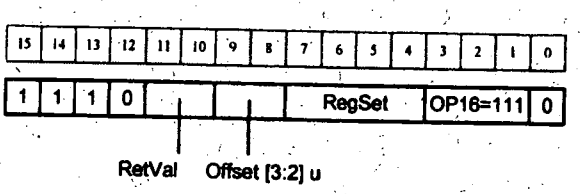



For details of the RegSet field see the pop and pop.ret instructions. The RegMask field has one bit for each of R1-R5, with the lowest bit selecting R1, through to the highest bit selecting R5.

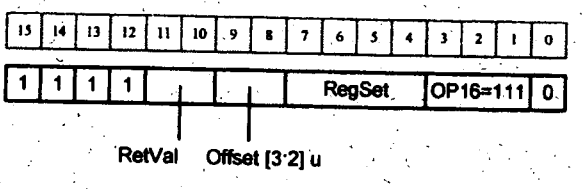
When RegSet is 0000 this represents the push instruction; all other values of RegSet represent the push.r instruction, with the register set comprising both the registers specified by RegMask and the registers specified from the decoding of RegSet.

### 8.11.9.4 OP16B7 pop and pop.ret Instructions

pop




pop.ret




The RetVal field specifies one of the following:

| *Retval:* | *Return Value:* |
|---|---|
| 00 | R0 |
| 01 | R1 |
| 10 | R6 |
| 11 | #1 |

The RegSet field specifies the following register sets:

| *RegSet:* | *Register Set:* |
|---|---|
| 0000 | {} |
| 0001 | {R14} |
| 0010 | {R6} |
| 0011 | {R6, R14} |
| 0100 | {R6, R7} |
| 0101 | {R6, R7, R14} |
| 0110 | {R6, R7, R8} |
| 0111 | {R6, R7, R8, R14} |
| 1000 | {R6, R7, R8, R9} |
| 1001 | {R6, R7, R8, R9, R14} |
| 1010 | {R6, R7, R8, R9, R10} |
| 1011 | {R6, R7, R8, R9, R10, R14} |
| 1100 | {R6, R7, R8, R9, R10, R11} |

| *RegSet:* | *Register Set:* |
|---|---|
| 1101 | {R6, R7, R8, R9, R10, R11, R14} |
| 1110 | {R6, R7, R8, R9, R10, R11, R12, R14} |
| 1111 | {R6, R7, R8, R9, R10, R11, R12, R13, R14} |

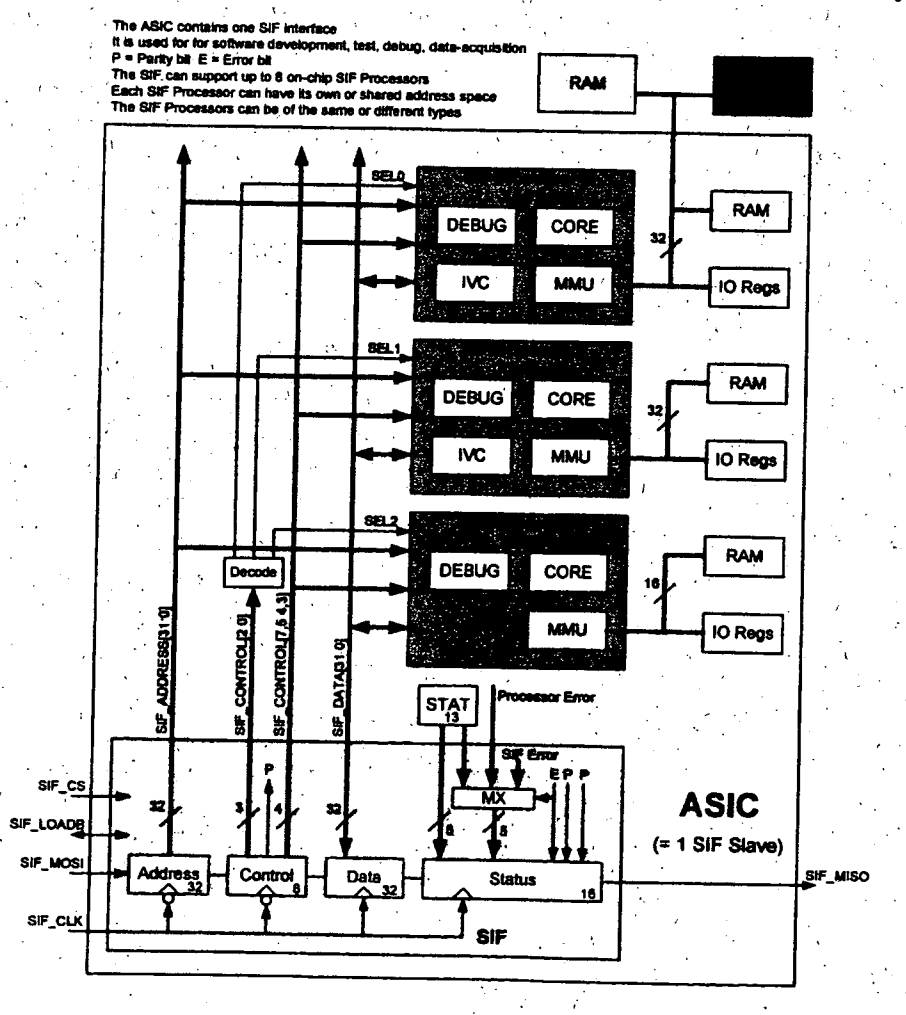
# 9 SIF Serial InterFace

A XAP3 ASIC is a single SIF Slave as defined in C4862-TM-005. The ASIC only contains one SIF interface (with 4 or 5 external pins). The default SIF provided with XAP3 can support up to 8 on-chip SIF Processors. These can all be XAP3 or can be a variety of different processors. Each can have its own or shared address space.

The SIF allows the user to debug software and to read or write the registers, memory and IO registers of all the on-chip SIF Processors. The SIF is also used for data acquisition and analogue production test of the ASIC. All memory and IO register reads and writes are performed in a non-invasive manner. The processors continue to operate in functional mode without having their timing affected.

## 9.1 SIF ASIC Block Diagram

Here is a block diagram of an ASIC that contains two XAP3a and one XAP2 processor, each with their own address space. It shows how the single SIF interface is used to access all 3 processors. The SIF_CS pin is not needed for a Single Slave SIF system.



## 9.2 SIF ASIC Modes

At any time, the ASIC has :

- One SIF Mode (because the ASIC only contains one SIF).
- A SIF Processor Mode for each Processor.

### *9.2.1 SIF Modes*

There are 2 SIF Modes; Normal and Command. These modes are properties of the SIF and are independent of the processor types or states. More information on these modes can be found in C4862-TM-005. For the purpose of readability, the 3 statements in each row are defined to be equivalent :

| *Full Statement* | *Computer Style* | *Shortened Statement* |
|---|---|---|
| The SIF is in Normal SIF Mode | SIF Mode = Normal | The SIF is in Normal mode |
| The SIF is in Command SIF Mode | SIF Mode = Command | The SIF is in Command mode |

#### 9.2.1.1 SIF Mode = Normal

The ASIC resets with SIF in Normal mode. The properties of Normal SIF Mode are :

- SIF Operation is called a SIF Instruction.
- The SIF shift register length can be different from one ASIC design to another. This allows the Address, Control, Data and Status fields to have different lengths for different ASIC designs. The default shift register lengths are; XAP1 = 36 bits, XAP2 = 64 bits, XAP3 = 88 bits, XAP4 = 52 bits. The rest of this document assumes that an 88-bit SIF shift register is being used.
- SIF can access all the on-chip processors and their associated memory spaces.
- SIF can execute Direct SIF Instructions. These are executed within the SIF itself and do not access a processor. Direct SIF Instructions require the DEBUG bit to be 1 and the PROCESSOR bits to be 0.

#### 9.2.1.2 SIF Mode = Command

The SIF Master can issue a special sequence of 256 bits on SIF_MOSI to put the SIF into Command mode. The properties of Command SIF Mode are :

- SIF Operation is called a SIF Command.
- It uses an 8-bit shift register. This length is the same for all SIF devices.

- Commands are 8 bits in to SIF_MOSI.
- Responses are 8 bits out from SIF_MISO.
- It is used to query or configure the SIF.
- It cannot access the processors or their associated memory spaces.
- It is used to reveal the hardware details of a particular SIF Slave (field lengths for Address, Control, Data, Status, Polarity of Write bit, etc).
- It is used by the SIF Master (e.g xIDE) at the beginning of a SIF communication. This allows the SIF Master to automatically discover the SIF devices without needing any manual configuration.
- Once the SIF Master has discovered all that it needs to know about the SIF Slave, it will put the SIF back into Normal mode. It will normally remain in Normal mode for the rest of the SIF communication.

### *9.2.2 XAP3 SIF Processor Modes*

Most processors have 2 SIF Processor Modes; Normal and Debug. At any time each processor has its own SIF Processor Mode. Each processor's mode can only be changed by the SIF interface.

The SIF Processor can be a variety of different processors. This specification describes the behaviour of the XAP3 in Normal and Debug SIF Processor Modes. For the purpose of readability, the 3 statements in each row are defined to be equivalent :

| *Full Statement* | *Computer Style* | *Shortened Statement* |
|---|---|---|
| The XAP3 is in Normal SIF Processor Mode | XAP3 SIF Processor Mode = Normal | The XAP3 is in Normal mode |
| The XAP3 is in Debug SIF Processor Mode | XAP3 SIF Processor Mode = Debug | The XAP3 is in Debug mode |

#### 9.2.2.1 XAP3 SIF Processor Mode = Normal

The ASIC resets with all XAP3 processors in Normal mode. This allows the SIF Master to read and write address-mapped variables (normally IO registers or memory) of the selected XAP3. These can be 8, 16 or 32-bit.

If the XAP3 is in Normal mode, then it will be running. It can only be stopped when in Debug mode.

#### 9.2.2.2 XAP3 SIF Processor Mode = Debug

The selected XAP3 is put into Debug mode by executing the 'Debug Enable' SIF Instruction. It will then remain in Debug mode until it executes the 'Debug Disable' SIF Instruction (when it will return to Normal mode).

Some SIF Instructions can only be executed if the XAP3 is already in Debug mode. In practise, Debug mode is only used during software development and ASIC test. Debug mode allows the user to perform functions such as start, stop, set breakpoint, run to breakpoint, read and write XAP3 internal registers and flags, etc.

It is important to distinguish between (Normal/Debug SIF Instructions) and (Normal/Debug XAP3 SIF Processor Modes) :

- A Normal SIF Instruction is when the DEBUG bit (bit 35 of SIF shift register) = 0. This is for 8, 16 and 32-bit reads and writes to any memory address.
- A Debug SIF Instruction is when the DEBUG bit (bit 35 of SIF shift register) = 1. This is for all SIF instructions apart from the reads and writes to memory.
- Only certain SIF Instructions (Normal Instructions + a few Debug Instructions) can be executed when the XAP3 is in Normal mode. The XAP3 is in Normal mode after reset and after the 'Debug Diasable' SIF Instruction.
- All SIF Instructions can be executed when the XAP3 is in Debug Mode. The XAP3 is in Debug Mode after the 'Debug Enable' SIF Instruction.

## 9.3 When are SIF Operations executed ?

### 9.3.1 SIF Mode = Command

When the SIF is in Command Mode, SIF Operations (= SIF Commands) are executed immediately. As soon as the 8-bit Command has been shifted in to SIF_MOSI, the 8-bit Response will shift out on SIF_MISO. There is never a wait period. SIF_LOADB is not used. This is why SIF Commands cannot access processors or memory space. They are normally only used to query the SIF configuration (i.e to read a small ROM inside the SIF).

### 9.3.2 SIF Mode = Normal

When the SIF is in Normal mode, the SIF Operation (= SIF Instruction) consists of 3 stages as follows :

1. **Shift In Stage**. SIF Master shifts SIF Instruction (Address/Control for Read, Address/Control/Data for Write) into SIF Slave with SIF_CLK and SIF_MOSI.
2. **Handshake Stage**. SIF Master pulls SIF_LOADB low for a short pulse. SIF Slave holds SIF_LOADB low until the SIF Instruction has been completed. The SIF passes the SIF Instruction on to the selected SIF Processor (e.g XAP3 identified by CONTROL[2:0]). When/if the selected processor completes the SIF Instruction, it tells the SIF( Slave) which lets go of SIF_LOADB. SIF_LOADB goes high, telling the SIF Master that the SIF Instruction has been completed. Sometimes the handshake is not completed, which is called a SIF Lockup.
3. **Shift Out Stage**. SIF Master shifts the result (Data/Status for Read, Status for Write) out of SIF Slave with SIF_CLK and SIF_MISO.

In the second stage (Handshake) the SIF Slave holds SIF_LOADB low for varying amounts of time. This depends upon the type of SIF Instruction and whether the selected XAP3 is stopped or running. As soon as the selected XAP3 executes the SIF Instruction, it tells the SIF (Slave). The SIF can then let go of SIF_LOADB, allowing it to return high again. The same handshake process is used for Direct SIF Instructions (which do not access any processor).

#### 9.3.2.1 Immediate SIF Handshake

In some cases the selected XAP3 executes the SIF Instruction immediately :

- All SIF Instructions if the selected XAP3 is currently executing a 'sleepsif' instruction (in Normal or Debug Mode, Stopped or Running). This implies that the XAP3 is waiting for a 'wake_up' hardware input (to move the processor on to its next instruction).
- All SIF Instructions if the selected XAP3 is stopped (which can only happen in Debug Mode).
- Debug SIF Instructions where (Address = 0x0 to 0xFF). This includes Debug Enable, Debug Disable, Start, Stop etc. (see XAP3 SIF Instruction table for details). Such instructions are executed immediately even if the selected XAP3 is running.
- Direct SIF Instructions.

The following SIF Instructions are error conditions which are recognised by the hardware (selected processor or SIF itself). The hardware will respond immediately, returning the appropriate Processor or SIF error code :

- Parity error in PARITY_AC
- Undefined SIF Instruction (Combination of Control and Address fields).
- Wrong data-value for SIF Instructions that require special data-values.
- SIF Write to a read-only register.
- SIF Read from a write-only register.
- Debug SIF Instruction which requires the selected XAP3 to be in Debug mode, when it is actually in Normal mode.
- Debug SIF Instruction which requires the selected XAP3 to be stopped, when it is actually running.

#### 9.3.2.2 Delayed SIF Handshake

In most cases the selected XAP3 waits for a 'sif' or 'sleepsif' XAP3 instruction before executing the SIF Instruction :

- All Normal SIF Instructions when the XAP3 is running (Normal or Debug mode).
- All Debug SIF Instructions where (Address > 0xFF) and when the XAP3 is running (Normal or Debug Mode).

#### 9.3.2.3 Incomplete SIF Handshake (ending in SIF Lockup)

The following SIF Instructions are error conditions which cannot be recognised by the hardware (selected processor or SIF itself). In such cases the SIF Slave will permanently hold SIF_LOADB low :

- Normal SIF Instruction when the selected XAP3 is running but never executes a 'sif' or 'sleepsif' instruction.

In such cases the selected XAP3 will never tell the SIF that it has completed the SIF Instruction, so the SIF Slave will never release SIF_LOADB, which the SIF Master will

see (and will know that this is being done by the selected SIF Slave). This has the effect of locking up the SIF Interface. The SIF Master must clear this with a SIF Cancel before it can proceed with any further SIF Instructions.

#### 9.3.2.4 SIF Cancel (to clear SIF Lockup)

The SIF has locked up because the SIF Master issued an invalid SIF Instruction to the selected SIF Processor in the selected SIF Slave. The SIF Master must be able to detect and recover from such a situation by :

- Having a timeout if SIF_LOADB stays low for too long.
- Issueing a SIF Cancel to clear the lockup in the SIF Slave (or Slaves)

There are 2 types of SIF Cancel (both described in more detail in C4862-TM-005) :

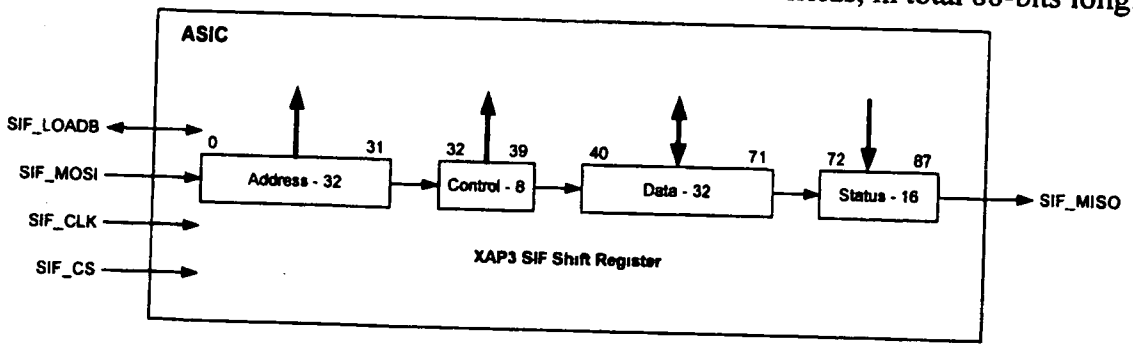
- SIF Master toggles SIF_CLK 32 times (this will clear all Slaves that are locked up).
- SIF Master pulls SIF_CS low (not implemented in SIF Slaves before 2005).

In both cases, the SIF Slave will release SIF_LOADB, allowing it to go high again. At this point the SIF lockup has been cleared. This allows the SIF Master to issue further SIF Instructions.

It is possible for the requested SIF Instruction (debug or normal mode, read or write) to be completed even after the Master has started a SIF Cancel. This means that when a Master issues a SIF Cancel, it cannot know whether the previously requested SIF Instruction will happen or not. The Master can assume that if the previous SIF instruction was executed by the Slave, then it will be the correct one requested.

## 9.4 SIF Shift Register

XAP3 ASICs contain a SIF shift register which consists of 4 fields, in total 88-bits long:




### *9.4.1 Address Field*

The 32-bit Address field is interpreted as follows for different SIF Instructions :

- Normal SIF Instructions – Address to read or write in units of the selected processor (e.g bytes or 16-bit words or 32-bit words).
- Debug SIF Instructions – OpCode for a particular operation

### *9.4.2 Control Field*

The 8-bit Control field, CONTROL[7:0] consists of :

- {WRITE, PARITY_AC, SIZE[1:0], DEBUG, PROCESSOR[2:0]}.

All SIF transactions are one of :

- SIF Read
- SIF Write

The WRITE bit is 1 for SIF Write Instructions and 0 for SIF Read Instructions. For all modes, the only way that the SIF can update any part of the ASIC is with a SIF Write. SIF Reads cannot update any part of the ASIC.

The PARITY_AC bit is set by the SIF Master and checked by the SIF Slave. If the Slave finds a parity error it should set ERROR, ERROR_TYPE and ERROR_CODE[3:0] in the Status field.

The 2 SIZE bits indicate whether the data is 8, 16 or 32-bit. The SIZE bits are interpreted for all Normal SIF Instructions (i.e reads and writes to memory and IO registers), whether the selected XAP3 is in Normal or Debug mode. Debug SIF Instructions which have a Data parameter must set the SIZE bits correctly. Debug SIF Instructions which do not have a Data parameter must set the SIZE bits to 0.

The DEBUG bit is 1 for Debug SIF Instructions and 0 for Normal SIF Instructions.

The 3 PROCESSOR bits select which of the 8 SIF Processors the SIF Instruction should be passed to. All SIF Instructions must select a SIF Processor. Only One SIF Processor can be accessed at a time. There is no state. Successive SIF Instructions can access any SIF Processors in any order. SIF Processor Addresses should be allocated from 0 upwards. If the ASIC only contains one XAP3, it should be at SIF Processor Address 0.

### 9.4.3 Data Field

The 32-bit Data field is used for SIF Reads and Writes (from or to the specified Memory Address or Register).

For security reasons, some Debug SIF Instructions require the Data field to be set to a specific value.

The Slave should only update the Data field after valid SIF Reads. If there has been any kind of error (i.e ERROR = 1), the Data field should not be updated.

### 9.4.4 Status Field

The 16-Bit Status field, STATUS[15:0] format depends upon whether there has been an error in the SIF Instruction or not. The Slave SIF has a 13-bit input from the main part of the ASIC called STAT[12:0].

The format of the Status field is independent of the specified Address and Control fields. Its format depends upon STAT[12:0] and on whether there was a SIF Error.

Under the various conditions, the SIF should format the Status field as follows :

| *STATUS Bits* | *STATUS Description* | *No Error* | *SIF Error* | *Processor Error* |
|---|---|---|---|---|
| 7:0 | STATUS[7:0] | STAT[7:0] | STAT[7:0] | STAT[7:0] |
| 11:8 | ERROR_CODE[3:0] | STAT[11:8] | SIF Error Code | Proc Error Code |
| 12 | ERROR_TYPE | STAT[12] | 0 | 1 |
| 13 | ERROR | 0 | 1 | 1 |
| 14 | PARITY_D | Data Parity | Data Parity | Data Parity |
| 15 | PARITY_S | Status Parity | Status Parity | Status Parity |

PARITY_D is set first, based on DATA[31:0].

PARITY_S is then set based on STATUS[14:0].

The Slave should update the Status field after all completed (SIF_LOADB handshake) SIF Instructions. This should happen whether there is an error or not.

### 9.4.5 Shift Register Bit Definitions

The XAP3 SIF shift register is defined as follows :

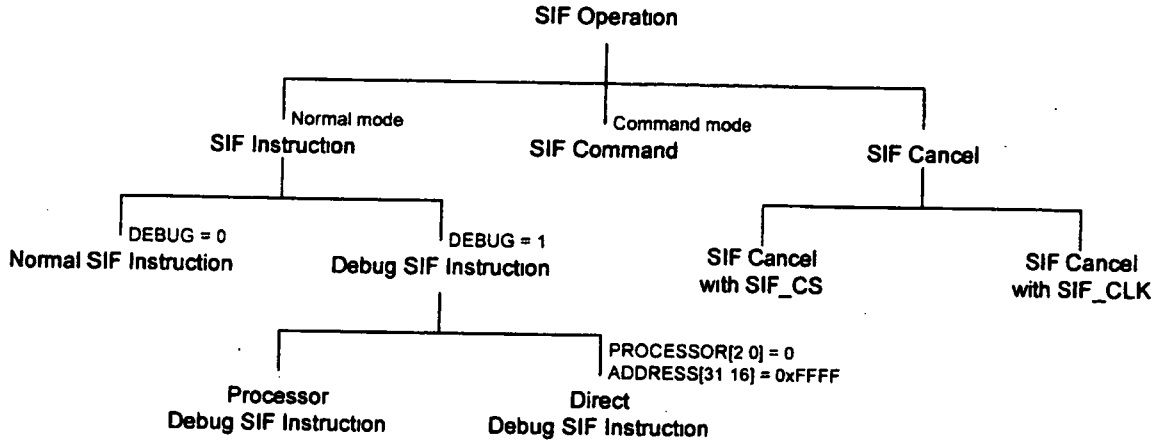
| *Bit* | *Name* | *Comments* |
|---|---|---|
| 0 | ADDRESS[0] | Memory address for Normal SIF Instructions. Address is in units of selected SIF Processor (i.e bytes for XAP3, 16-bit words for XAP2 or XAP1).<br><br>Parameter for Debug SIF Instructions (see SIF Instruction Table). |
| ... | | |
| 31 | ADDRESS[31] | |
| 32 | PROCESSOR[0] | PROCESSOR[2:0] = CONTROL[2:0].<br>Selects which SIF Processor (out of 8) to use.<br>SIF Processor addresses should start from 0 and work up to 7. |
| 33 | PROCESSOR[1] | |
| 34 | PROCESSOR[2] | |
| 35 | DEBUG | DEBUG = CONTROL[3].<br>0 = Normal SIF Instruction<br>1 = Debug SIF Instruction |
| 36 | SIZE[0] | SIZE[1:0] = CONTROL[5:4].<br>Determines the data length of the SIF access.<br>00 = 8-bit, 01 = 16-bit,10 = 32-bit, 11 = reserved |
| 37 | SIZE[1] | |
| 38 | PARITY_AC | PARITY_AC = CONTROL[6].<br>Set by Master to be odd parity (odd number of 1s) with :<br>• CONTROL[7]<br>• CONTROL[5:0]<br>• ADDRESS[31:0] |
| 39 | WRITE | WRITE = CONTROL[7].<br>0 = Read, 1 = Write<br>*Note that this is the opposite polarity from XAP1 and XAP2. The SIF_MOSI pad is tied low, so in XAP3 a random SIF transaction will cause a read and not a write.* |

| | | |
|---|---|---|
| 40 | DATA[0] | 32-bit: DATA[31:0] is written to or read from the selected address.<br><br>16-bit Write: DATA[15:0] is written to selected address.<br>16-bit Read: DATA[31:16] = 0, DATA[15:0] = data read from selected address.<br><br>8-bit Write: DATA[7:0] is written to selected address.<br>8-bit Read: DATA[31:8] = 0, DATA[7:0] = data read from selected address. |
| ... | | |
| 71 | DATA[31] | |
| 72 | STATUS[0] | After all completed SIF Instructions, STATUS[7:0] is updated with the hardware status word (STAT[7:0]), independent of the selected Address and Control fields.<br><br>This will often be used for a TimeStamp, where STAT[0] is the highest frequency bit. |
| ... | | |
| 79 | STATUS[7] | |
| 80 | ERROR_CODE[0] | STATUS[11:8] is updated with :<br>• STAT[11:8], if there is no error.<br>• SIF Error Code, if there is a SIF error.<br>• Processor Error Code, if there is a Processor error. |
| 81 | ERROR_CODE[1] | |
| 82 | ERROR_CODE[2] | |
| 83 | ERROR_CODE[3] | |
| 84 | ERROR_TYPE | STATUS[12] is updated with :<br>• STAT[12], if there is no error.<br>• 0, if there is a SIF error.<br>• 1, if there is a Processor error. |
| 85 | ERROR | ERROR = STATUS[13].<br>• 0 = No error. SIF Instruction completed successfully.<br>• 1 = Error. SIF or selected Processor detected an error in the requested SIF Instruction. |
| 86 | PARITY_D | PARITY_D = STATUS[14]<br><br>Set by Slave to be odd parity (odd number of 1s) with:<br>• DATA[31:0]<br><br>This is set before PARITY_S. |
| 87 | PARITY_S | PARITY_S = STATUS[15].<br><br>Set by Slave to be odd parity (odd number of 1s) with:<br>• STATUS[14:0]<br><br>This is set after PARITY_D. |

## 9.5 SIF Instructions

### *9.5.1 SIF Instruction Types*

SIF Operations can be categorised as follows :




XAP3 SIF instructions are defined by the Control and Address shift register fields.

There are 2 types of SIF Instruction, defined as follows :

- Normal SIF Instructions DEBUG = 0
- Debug SIF Instructions DEBUG = 1

A particular kind of Debug SIF Instruction is the Direct SIF Instruction. These are executed in the SIF itself and do not access any processor. Direct SIF Instructions all format the shift register bits as follows :

- DEBUG = 1
- PROCESSOR[2:0] = 0
- ADDRESS [31:16] = 0xFFFF

All other Debug Instructions refer to the selected processor. Such Debug SIF Instructions can be split into 3 categories :

- Address < 0x80 : No mode requirements.
- 0x7F < Address < 0x100 Selected XAP3 must be in Debug mode.
- 0xFF < Address < 0x1000 Selected XAP3 must be in Debug mode and stopped.

Like the XAP3 itself, SIF reads and writes can use any byte address for all word sizes. The words do not need to be memory-aligned. i.e :

- Addresses for 32-bit words do not need to be a multiple of 4
- Addresses for 16-bit words do not need to be a multiple of 2

### 9.5.2 SIF Instruction Table

The following XAP3 SIF Instructions (Normal then Direct then other Debug) cover a subset of possible values for the Control and Address fields. The table does not include the PARITY_AC or WRITE bits in the Control field. The table assumes that PROCESSOR[2:0] = 0. All unspecified values are reserved for future use.

| *Control [5:0]* | *Address* | *Data* | *Required XAP3 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x00 | 0xXXXXXXXX | 8-bit | - | RW | Byte Memory Read/Write |
| 0x10 | 0xXXXXXXXX | 16-bit | - | RW | 16-bit Memory Read/Write |
| 0x20 | 0xXXXXXXXX | 32-bit | - | RW | 32-bit Memory Read/Write |
| 0x28 | 0xFFFF0000 | 32-bit | - | R | SIF Read Counter |
| 0x28 | 0xFFFF0001 | 32-bit | - | R | SIF Write Counter |
| 0x28 | 0xFFFF0002 | 32-bit | - | R | SIF Error Counter |
| 0x28 | 0xFFFF0003 | 32-bit | - | R | SIF Cancel Counter |
| 0x28 | 0xFFFF0100 | 0xD5IF | - | W | Reset Everything<br>• Off-Chip Devices<br>• ASIC |
| 0x28 | 0xFFFF0101 | 0xD5IF | - | W | Debug Reset Everything<br>• Off-Chip Devices<br>• ASIC<br>• Puts all XAP3s into Debug mode and Stopped |
| 0x28 | 0xFFFF0110 | 0xD5IF | - | W | Reset Off-Chip Devices |
| 0x28 | 0xFFFF0120 | 0xD5IF | - | W | Reset ASIC |
| 0x28 | 0xFFFF0121 | 0xD5IF | - | W | Debug Reset ASIC<br>• Puts all XAP3s into Debug mode and Stopped |
| 0x28 | 0xFFFF0130 | 0xD5IF | - | W | Reset SIF |
| 0x28 | 0x0 | 0xD0FF | - | W | Debug Disable<br>• Puts XAP3 into Normal mode<br>• Sets XAP3 Running |
| 0x28 | 0x1 | 0xDBE | - | W | Debug Enable<br>• Puts XAP3 into Debug mode |
| 0x28 | 0x2 | 32-bit | - | R | Status Read |

| *Control [5:0]* | *Address* | *Data* | *Required XAP3 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x28 | 0x3 | 32-bit | - | R | Version Number |
| 0x28 | 0x4 | 32-bit | - | R | Licence Number |
| 0x28 | 0x5 | 32-bit | - | R | Register PC |
| 0x28 | 0x6 | 32-bit | - | R | Last PC<br>• Address of last completed instruction |
| 0x08 | 0x81 | - | Debug | W | Stop |
| 0x08 | 0x82 | - | Debug | W | Run |
| 0x08 | 0x83 | - | Debug | W | Wakeup |
| 0x08 | 0x84 | - | Debug | W | Single Step |
| 0x08 | 0x85 | - | Debug | W | Run to Break |
| 0x08 | 0x91 | - | Debug | W | Debug Reset selected XAP3<br>• Puts XAP3 into Debug mode and Stopped |
| 0x08 | 0x92 | - | Debug | W | Reset selected User Gates |
| 0x28 | 0x100 | 32-bit | - | R | Register R0 |
| 0x28 | 0x101 | 32-bit | - | R | Register R1 current mode |
| 0x28 | 0x101 | 32-bit | Debug Stopped | W | Register R1 current mode |
| 0x28 | 0x102 | 32-bit | - | R | Register R2 current mode |
| 0x28 | 0x102 | 32-bit | Debug Stopped | W | Register R2 current mode |
| 0x28 | 0x103 | 32-bit | - | R | Register R3 |
| 0x28 | 0x103 | 32-bit | Debug Stopped | W | Register R3 |
| 0x28 | 0x104 | 32-bit | - | R | Register R4 |
| 0x28 | 0x104 | 32-bit | Debug Stopped | W | Register R4 |
| 0x28 | 0x105 | 32-bit | - | R | Register R5 |
| 0x28 | 0x105 | 32-bit | Debug Stopped | W | Register R5 |
| 0x28 | 0x106 | 32-bit | - | R | Register R6 |
| 0x28 | 0x106 | 32-bit | Debug Stopped | W | Register R6 |
| 0x28 | 0x107 | 32-bit | - | R | Register R7 |
| 0x28 | 0x107 | 32-bit | Debug Stopped | W | Register R7 |
| 0x28 | 0x108 | 32-bit | - | R | Register R8 |

| *Control [5:0]* | *Address* | *Data* | *Required XAP3 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x28 | 0x108 | 32-bit | Debug Stopped | W | Register R8 |
| 0x28 | 0x109 | 32-bit | - | R | Register R9 |
| 0x28 | 0x109 | 32-bit | Debug Stopped | W | Register R9 |
| 0x28 | 0x10A | 32-bit | - | R | Register R10 |
| 0x28 | 0x10A | 32-bit | Debug Stopped | W | Register R10 |
| 0x28 | 0x10B | 32-bit | - | R | Register R11 |
| 0x28 | 0x10B | 32-bit | Debug Stopped | W | Register R11 |
| 0x28 | 0x10C | 32-bit | - | R | Register R12 |
| 0x28 | 0x10C | 32-bit | Debug Stopped | W | Register R12 |
| 0x28 | 0x10D | 32-bit | - | R | Register R13 |
| 0x28 | 0x10D | 32-bit | Debug Stopped | W | Register R13 |
| 0x28 | 0x10E | 32-bit | - | R | Register R14 current mode |
| 0x28 | 0x10E | 32-bit | Debug Stopped | W | Register R14 current mode |
| 0x28 | 0x10F | 32-bit | - | R | Register R15 current mode |
| 0x28 | 0x10F | 32-bit | Debug Stopped | W | Register R15 current mode |
| 0x28 | 0x111 | 32-bit | - | R | Register R1_U |
| 0x28 | 0x111 | 32-bit | Debug Stopped | W | Register R1_U |
| 0x28 | 0x112 | 32-bit | - | R | Register R2_U |
| 0x28 | 0x112 | 32-bit | Debug Stopped | W | Register R2_U |
| 0x28 | 0x11E | 32-bit | - | R | Register R14_U |
| 0x28 | 0x11E | 32-bit | Debug Stopped | W | Register R14_U |
| 0x28 | 0x11F | 32-bit | - | R | Register R15_U |
| 0x28 | 0x11F | 32-bit | Debug Stopped | W | Register R15_U |
| 0x28 | 0x121 | 32-bit | - | R | Register R1_S |
| 0x28 | 0x121 | 32-bit | Debug Stopped | W | Register R1_S |
| 0x28 | 0x122 | 32-bit | - | R | Register R2_S |

| *Control [5:0]* | *Address* | *Data* | *Required XAP3 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x28 | 0x122 | 32-bit | Debug Stopped | W | Register R2_S |
| 0x28 | 0x12E | 32-bit | - | R | Register R14_S |
| 0x28 | 0x12E | 32-bit | Debug Stopped | W | Register R14_S |
| 0x28 | 0x12F | 32-bit | - | R | Register R15_S |
| 0x28 | 0x12F | 32-bit | Debug Stopped | W | Register R15_S |
| 0x28 | 0x131 | 32-bit | - | R | Register R1_I |
| 0x28 | 0x131 | 32-bit | Debug Stopped | W | Register R1_I |
| 0x28 | 0x132 | 32-bit | - | R | Register R2_I |
| 0x28 | 0x132 | 32-bit | Debug Stopped | W | Register R2_I |
| 0x28 | 0x13E | 32-bit | - | R | Register R14_I |
| 0x28 | 0x13E | 32-bit | Debug Stopped | W | Register R14_I |
| 0x28 | 0x13F | 32-bit | - | R | Register R15_I |
| 0x28 | 0x13F | 32-bit | Debug Stopped | W | Register R15_I |
| 0x28 | 0x141 | 32-bit | - | R | Register R1_N |
| 0x28 | 0x141 | 32-bit | Debug Stopped | W | Register R1_N |
| 0x28 | 0x14E | 32-bit | - | R | Register R14_N |
| 0x28 | 0x14E | 32-bit | Debug Stopped | W | Register R14_N |
| 0x28 | 0x200 | 32-bit | - | R | Register BRK0 |
| 0x28 | 0x200 | 32-bit | Debug Stopped | W | Register BRK0 |
| 0x28 | 0x201 | 32-bit | - | R | Register BRK1 |
| 0x28 | 0x201 | 32-bit | Debug Stopped | W | Register BRK1 |
| 0x28 | 0x202 | 32-bit | - | R | Register BRK2 |
| 0x28 | 0x202 | 32-bit | Debug Stopped | W | Register BRK2 |
| 0x28 | 0x203 | 32-bit | - | R | Register BRK3 |
| 0x28 | 0x203 | 32-bit | Debug Stopped | W | Register BRK3 |

| *Control [5:0]* | *Address* | *Data* | *Required XAP3 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x28 | 0x204 | 32-bit | - | R | Register BRK4 |
| 0x28 | 0x204 | 32-bit | Debug Stopped | W | Register BRK4 |
| 0x28 | 0x205 | 32-bit | - | R | Register BRK5 |
| 0x28 | 0x205 | 32-bit | Debug Stopped | W | Register BRK5 |
| 0x28 | 0x206 | 32-bit | - | R | Register BRK6 |
| 0x28 | 0x206 | 32-bit | Debug Stopped | W | Register BRK6 |
| 0x28 | 0x207 | 32-bit | - | R | Register BRK7 |
| 0x28 | 0x207 | 32-bit | Debug Stopped | W | Register BRK7 |
| 0x28 | 0x208 | 32-bit | - | R | Register BRK4COUNT |
| 0x28 | 0x208 | 32-bit | Debug Stopped | W | Register BRK4COUNT |
| 0x28 | 0x209 | 32-bit | - | R | Register BRK5COUNT |
| 0x28 | 0x209 | 32-bit | Debug Stopped | W | Register BRK5COUNT |
| 0x28 | 0x20A | 32-bit | - | R | Register BRK6COUNT |
| 0x28 | 0x20A | 32-bit | Debug Stopped | W | Register BRK6COUNT |
| 0x28 | 0x20B | 32-bit | - | R | Register BRK7COUNT |
| 0x28 | 0x20B | 32-bit | Debug Stopped | W | Register BRK7COUNT |
| 0x28 | 0x20C | 32-bit | - | R | Register BRK6MASK |
| 0x28 | 0x20C | 32-bit | Debug Stopped | W | Register BRK6MASK |
| 0x28 | 0x20D | 32-bit | - | R | Register BRK7MASK |
| 0x28 | 0x20D | 32-bit | Debug Stopped | W | Register BRK7MASK |
| 0x28 | 0x20E | 32-bit | - | R | Register BRK6DATA |
| 0x28 | 0x20E | 32-bit | Debug Stopped | W | Register BRK6DATA |
| 0x28 | 0x20F | 32-bit | - | R | Register BRK7DATA |
| 0x28 | 0x20F | 32-bit | Debug Stopped | W | Register BRK7DATA |
| 0x28 | 0x300 | 32-bit | - | R | Register FLAGS |

| *Control [5:0]* | *Address* | *Data* | *Required XAP3 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x28 | 0x300 | 32-bit | Debug Stopped | W | Register FLAGS |
| 0x28 | 0x301 | 32-bit | - | R | Register SFLAGS_S |
| 0x28 | 0x301 | 32-bit | Debug Stopped | W | Register SFLAGS_S |
| 0x28 | 0x302 | 32-bit | - | R | Register SFLAGS_I |
| 0x28 | 0x302 | 32-bit | Debug Stopped | W | Register SFLAGS_I |
| 0x28 | 0x303 | 32-bit | - | R | Register SFLAGS_N |
| 0x28 | 0x303 | 32-bit | Debug Stopped | W | Register SFLAGS_N |
| 0x28 | 0x304 | 32-bit | - | R | Register GP_U |
| 0x28 | 0x304 | 32-bit | Debug Stopped | W | Register GP_U |
| 0x28 | 0x305 | 32-bit | - | R | Register GP_S |
| 0x28 | 0x305 | 32-bit | Debug Stopped | W | Register GP_S |
| 0x28 | 0x306 | 32-bit | - | R | Register GP_I |
| 0x28 | 0x306 | 32-bit | Debug Stopped | W | Register GP_I |
| 0x28 | 0x307 | 32-bit | - | R | Register IVTB |
| 0x28 | 0x307 | 32-bit | Debug Stopped | W | Register IVTB |
| 0x28 | 0x308 | 32-bit | - | R | Register BRKE |
| 0x28 | 0x308 | 32-bit | Debug Stopped | W | Register BRKE |
| 0x28 | 0x314 | 32-bit | - | R | Register GP current mode |
| 0x28 | 0x314 | 32-bit | Debug Stopped | W | Register GP current mode |
| 0x28 | 0x400 | 32-bit | - | R | Register PC |
| 0x28 | 0x400 | 32-bit | Debug Stopped | W | Register PC |

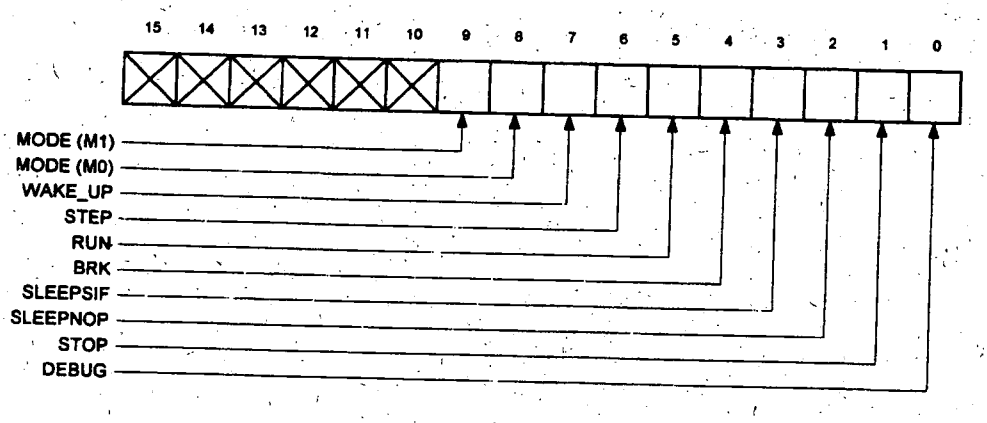
### 9.5.3 XAP3 Status Register

One of the SIF Debug Instructions reads the selected XAP3 Status Register. Note that this is not the same as the SIF shift register Status field, STATUS[15:0].

The XAP3 Status word contains the following information:

- Whether the selected XAP3 is in Debug or Normal mode.
- Whether the selected XAP3 program is running or stopped.
- Whether the selected XAP3 is awake or asleep.
- Current processor mode (User, Supervisor, Interrupt or NMI) of the selected XAP3.

The XAP3 Status Register contains the following bit fields :



## 9.6 SIF Error Processing

### *9.6.1 SIF Parity Bits*

Normally parity bits can only expose single bit errors. However, the XAP3 SIF has a parity-checking configuration which also detects stuck high and stuck low faults on SIF_MOSI and SIF_MISO.

#### 9.6.1.1 PARITY_AC

PARITY_AC (Address + Control field parity) is set by the SIF Master and checked by the SIF Slave. It has odd parity, so there are an odd number of 1s in the 40 bits :

- CONTROL[7:0]
- ADDRESS[31:0]

If SIF_MOSI is stuck at 1 or stuck at 0, there will be a parity error in PARITY_AC. This is because there is an even number of protected bits (including PARITY_AC) and odd parity checking. This means that the bits cannot all be the same value.

If the Slave detects a parity error in PARITY_AC it should :

- Not pass the SIF Instruction on to a processor.
- Set ERROR = 1 in Status field.
- Set ERROR_TYPE = 0 in the Status field.
- Format ERROR_CODE[3:0] in the Status field.
- Complete the SIF Handshake (SIF_LOADB).

#### 9.6.1.2 PARITY_D

PARITY_D (Data field parity) is set by the SIF Slave and checked by the SIF Computer. It has odd parity, so there are an odd number of 1s in the 33 bits :

- PARITY_D
- DATA[31:0]

PARITY_D is based on the values in the :

- Data Capture Register, for valid SIF Reads (i.e when Data Shift Register is updated with a parallel load from the Data Capture Register).
- Data Shift Register for SIF Writes and SIF Errors (i.e when Data Shift Register is not updated with a parallel load from the Data Capture Register).

#### 9.6.1.3 PARITY_S

PARITY_S (Status field parity) is set by the SIF Slave and checked by the SIF Computer. PARITY_S should be calculated after PARITY_D. It has odd parity, so there are an odd number of 1s in the 16 bits :

- STATUS[15:0]

If SIF_MISO is stuck at 1 or stuck at 0, there will be a parity error in PARITY_S. This is because there is an even number of protected bits (including PARITY_S) and odd parity checking. This means that the bits cannot all be the same value.

PARITY_S is based on the values in the Status Capture Register (which will always be parallel loaded into the Status Shift Register when a SIF Instruction is completed).

### *9.6.2 SIF Error Reporting in Status Field*

#### 9.6.2.1 SIF Error Codes

When the SIF detects an error in the SIF Instruction it should set :

- ERROR = 1
- ERROR_TYPE = 0
- ERROR_CODE[3:0] as follows :

| *SIF Error Code* | *Description* |
|---|---|
| 0 | Parity error in PARITY_AC |
| 1 | Unrecognised Direct SIF Instruction |
| 2 | Wrong data-value in Direct SIF Instruction |
| 3 | Unrecognised Processor |

#### 9.6.2.2 Processor Error Codes

When the selected XAP3 detects an error in the SIF Instruction it should set :

- ERROR = 1
- ERROR_TYPE = 1
- ERROR_CODE[3:0] as follows :

| *Processor Error Code* | *Description* |
|---|---|
| 0 | Unrecognised SIF Instruction |
| 1 | Wrong data-value in SIF Instruction |
| 2 | SIF Write to a read-only register |
| 3 | SIF Read from a write-only register |
| 4 | Processor not in Debug mode |
| 5 | Processor not Stopped |

### 9.6.3 SIF Counters

The SIF contains four 32-bit counters that can be read by Direct SIF Instructions :

- SIF Read Counter
- SIF Write Counter
- SIF Error Counter
- SIF Cancel Counter

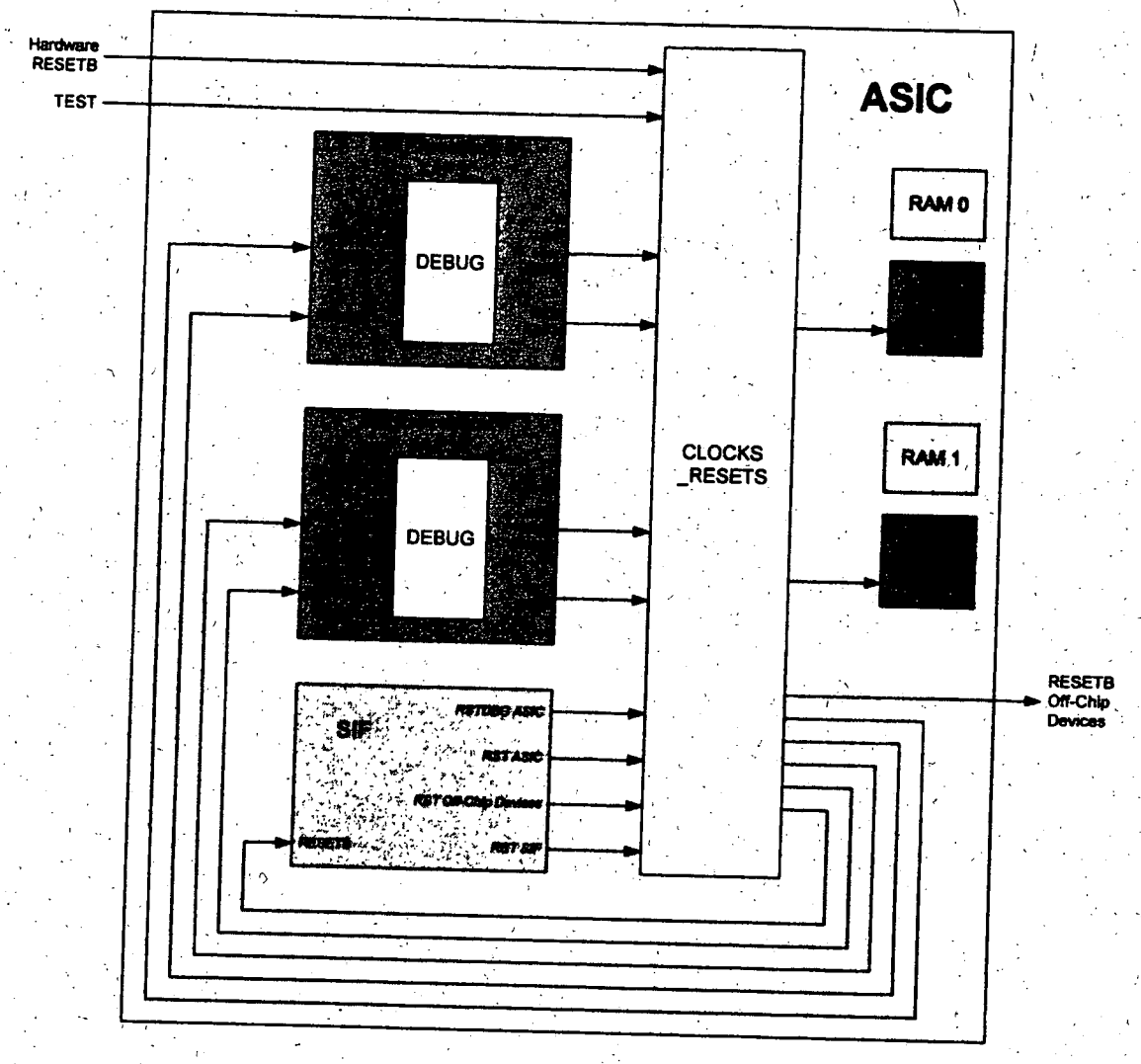
The counters reset to 0 (by hardware, ASIC or SIF Reset). The counters wrap and do not have any overflow detection.

The SIF increments each counter by 1 when the relevant event occurs :

- Read or Write or Error counter is incremented when the Capture registers are updated. 1 of these counters is incremented at the end of a completed SIF Instruction.
- Cancel counter is incremented when SIF_LOADB_SLAVE goes high as a result of a SIF Cancel.

It is possible for a SIF Read or Write to be completed after a SIF Cancel. In this case, the Cancel Counter and one of the other counters will be incremented.

## 9.7 SIF Resets




The SIF and XAP3 Verilog modules contain :

- RESETB* - Asynchronous active low reset inputs (all flipflops will be cleared even if clocks are stopped). By default, all flipflops are reset to 0.
- RST* - Synchronous (clocked by system CLK) active high reset outputs.

The SIF and XAP3 Verilog modules do not contain any test mode circuitry. They do not impose or assume any test strategy for the ASIC. They are scannable designs that the designer can configure for a variety of test strategies.

All conditioning of clock and reset signals as required for Scan and other test modes should be done in the CLOCKS_RESETS module. This module generates the RESETB restes that are fed to the various modules in the ASIC. These signals will often need to be controlled by external pins in TEST mode, to make the ASIC fully scannable. The

CLOCKS_RESETS module is application-specific and should be designed by the user to conform to the test strategy being used for the ASIC.

The SIF Instructions include several resets for different parts of the ASIC and Off-Chip devices. SIF resets are done by 2 mechanisms :

- Direct SIF Instructions. Reset 'everything', off-chip devices, whole ASIC, SIF itself.
- Processor SIF Instructions to a selected processor. Reset that processor, associated user gates

To avoid resets occuring by accident :

- Direct SIF Instruction resets require special data values.
- Processor SIF Instructions require the selected processor to be in Debug mode.

The XAP3 has 2 reset inputs :

- RESETB. Resets to the state defined section 5.9. i.e in Normal mode and Running.
- RESETB_DEBUG. Same as RESETB except that it is in Debug mode and Stopped.

RESETB_DEBUG is needed so that the whole ASIC or selected processors can be put into reset state and stopped, before single-stepping for software-debug.

-400-

Cambridge Consultants

# XAP3

## Programmer's Manual

**Cambridge Consultants Limited**
Science Park
Milton Road
Cambridge
England
CB4 0DW

+44 (0) 1223 420024

xap@cambridgeconsultants.com

**C7245-UM-002 v1.0**

12 May 2005

Cambridge Consultants

-401-

# Revision History

| Revision | Date | Details |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |

Cambridge Consultants

-402-

# Legal Notices

This is the Programmer's Manual for the XAP3 processor. You may use this if you accept the following conditions. If you do not accept these conditions, you must delete or dispose of this copy of the XAP3 Programmer's Manual immediately.

Copying You are free to copy this manual within your organisation, but you must not copy it in whole or in part to anyone outside your organisation without the prior written agreement of Cambridge Consultants Ltd. This manual can be downloaded from www.CambridgeConsultants.com/ASIC by those who accept these conditions

Fit for purpose The XAP3 processor and the xIDE software development environment must not be used for safety critical and/or life support applications without a specific written agreement from Cambridge Consultants Ltd

Liability Cambridge Consultants Ltd makes no warranty of any kind with regard to the information contained in this manual, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose Cambridge Consultants Ltd. shall not be liable for omissions or errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material. The information contained herein may be updated from time to time.

Ownership and Licence The XAP3 processor (including associated technologies), XAP3 documentation and the xIDE software development environment and all intellectual property rights associated with them are owned by Cambridge Consultants Ltd and are protected by copyright law and international copyright treaties; patent applications, letters patent and other intellectual property laws and treaties You must not attempt to use the XAP3 processor or the xIDE developer's environment unless you have a valid licence agreement with Cambridge Consultants Ltd You must not use the information within this manual or other items supplied to you for the purposes of designing, developing or testing a device or simulator which is, or is intended to be, wholly or partly instruction set compatible with the XAP3 processor. You are authorised to open, view and print any electronic copy we send you of this manual within your organisation. Printouts of this manual must be kept within your organisation

Copyright and trademarks This manual is © copyright Cambridge Consultants 2005. Cambridge Consultants Ltd , the Cambridge Consultants logo, XAP, XAP3 and xIDE are trademarks of Cambridge Consultants Ltd. All other trademarks mentioned herein are the property of their respective owners.

Cambridge Consultants Ltd
Science Park
Milton Rd
Cambridge CB4 0DW
England

www.CambridgeConsultants.com/ASIC

© Cambridge Consultants Ltd 2005

# Project Team

# Table of Contents


1 INTRODUCTION ..... 6

1.1 THIS DOCUMENT ..... 6
1.2 ASSUMPTIONS ..... 6
1.3 ABOUT XAP3 ..... 6
1.4 THE XAP FAMILY ..... 8
1.5 XAP3 SYSTEMS ..... 8
1.6 OTHER DOCUMENTATION ..... 9
1.7 FOR FURTHER HELP ..... 11
1.8 GLOSSARY AND ACRONYMS ..... 11

2 PROGRAMMER'S MODEL ..... 12

2.1 PROCESSOR STATES ..... 12
2.2 PROCESSOR MODES ..... 13
*2.2.1 Capabilities of the processor modes ..... 14*
*2.2.2 Typical usage of processor modes ..... 15*
2.3 REGISTERS ..... 15
*2.3.1 Normal registers ..... 15*
*2.3.2 Program Counter ..... 17*
*2.3.3 Special registers ..... 18*
*2.3.4 Breakpoint registers ..... 20*
2.4 PIPELINE ..... 21
2.5 MEMORY ARCHITECTURE ..... 21
2.6 STACK OPERATION ..... 22
2.7 RESET ..... 22
2.8 INTERRUPTS AND EXCEPTIONS ..... 22
*2.8.1 Exception processing in User Mode ..... 23*
*2.8.2 Exception processing in the privileged modes ..... 23*
*2.8.3 Interrupt processing ..... 24*
*2.8.4 Returning from interrupts and exceptions ..... 25*
*2.8.5 Nested interrupts ..... 25*
*2.8.6 Interrupt vector table ..... 25*
*2.8.7 Exceptions ..... 26*
2.9 DEBUGGING ..... 30

3 C/C++ LANGUAGE INTERFACE ..... 33

3.1 DATA TYPES ..... 33
3.2 DATA ALIGNMENT ..... 33
*3.2.1 Aligned data ..... 34*
*3.2.2 Unaligned data ..... 34*
*3.2.3 Methods of accessing unaligned data ..... 34*
3.3 CALLING CONVENTION ..... 35
3.4 APPLICATION MODELS ..... 36
*3.4.1 Static linkage ..... 36*
*3.4.2 Dynamc linkage ..... 36*

4 INSTRUCTION SET OVERVIEW ..... 37

4.1 SUMMARY OF ASSEMBLER SYNTAX ..... 37
*4.1.1 Instruction mnemonics ..... 37*
*4.1.2 Operands ..... 37*
*4.1.3 Registers ..... 38*
*4.1.4 Register Lists ..... 38*
*4.1.5 Comments ..... 38*

4.1.6 Number formats ..... 38
4.1.7 Labels ..... 39
4.1.8 Directives ..... 39
4.2 IMMEDIATE ENCODING ..... 39
4.2.1 .i instructions – no immediate encoding ..... 40
4.2.2 .i instructions – "s" encoding ..... 40
4.2.3 .i instructions – "u" encoding ..... 42
4.2.4 .h instructions – no immediate encoding ..... 42
4.2.5 .h instructions – "h" encoding ..... 42
4.2.6 .p instructions – "s" encoding ..... 43
4.2.7 .a instructions – "u" encoding ..... 43
4.2.8 mov.g – "u" encoding ..... 44
4.2.9 Register or small immediate values ..... 44
4.2.10 Examples of invalid immediates in instructions ..... 44
4.3 INSTRUCTION ENCODING ..... 45
4.4 ADDRESSING MODES ..... 45

**5 INSTRUCTION GROUPS ..... 47**

5.1.1 Branches ..... 47
5.1.2 Load and Store ..... 47
5.1.3 Push and Pop ..... 49
5.1.4 Move ..... 54
5.1.5 ALU operations ..... 55
5.1.6 Compare operations ..... 56
5.1.7 Shift and rotate ..... 56
5.1.8 DSP instructions ..... 56
5.1.9 Miscellaneous instructions ..... 57
5.2 COMMON CODE SEQUENCES ..... 58
5.2.1 Function prologue and epilogue ..... 58
5.2.2 Nested interrupt prologue and epilogue ..... 58
5.2.3 Semaphore ..... 59
5.2.4 Initialisation of User Mode registers ..... 60
5.2.5 DSP Example ..... 60
5.3 INSTRUCTIONS GROUPED BY FUNCTION ..... 61
5.3.1 Branches ..... 61
5.3.2 Load and Store ..... 62
5.3.3 Push and Pop ..... 62
5.3.4 Move ..... 63
5.3.5 Conditional Set ..... 63
5.3.6 ALU operations ..... 64
5.3.7 Compare operations ..... 65
5.3.8 Shift and rotate ..... 65
5.3.9 DSP instructions ..... 65
5.3.10 Miscellaneous instructions ..... 66
5.4 ALPHABETICAL LIST OF ALL INSTRUCTIONS ..... 66
5.5 PRIVILEGED INSTRUCTIONS ..... 73
5.6 ALIASED INSTRUCTIONS ..... 73
5.7 REGISTER NAMING IN INSTRUCTIONS ..... 74
5.8 REGISTER SPECIFICATION FIELDS ..... 74

**6 INSTRUCTION SET REFERENCE ..... 77**

# 1 Introduction

## 1.1 This Document

This document is the primary reference for software engineers developing programs to run on XAP3 systems. It contains all the information needed to understand the XAP3 architecture from a software perspective.

Further documents describe other aspects of XAP3 systems. Refer to section "Other documentation" for details.

## 1.2 Assumptions

The reader is assumed to be familiar microprocessor architectures in general, and to have some understanding of high level languages such as 'C'. A detailed understanding of digital electronics is not required.

## 1.3 About XAP3

The XAP3 is a powerful 32-bit processor with a von Neumann architecture. The processor can address 4GByte of memory, organised as a linear byte-addressable memory.

### *High Code Density*

The XAP3 architecture, instruction set, compiler toolchain and application binary interface are all designed for efficient execution of programs written in 'C' or C++. The code density achieved with the xap3ncc compiler matches industry benchmarks.

### *Complex Systems*

The XAP3 implements the features necessary to support complex software systems where high reliability or high integrity is a requirement. The XAP3 provides hardware support for common real time operating system primitives, including a clear definition of user and

supervisor modes, atomic instructions for synchronisation constructs, and position independent code to allow dynamic linkage.

The XAP3 MMU implements memory protection systems for instruction and data, throwing exceptions if processes access code or data memory illegally. These exceptions can also be used to implement virtual memory systems.

### Interrupts and Exceptions

The XAP3 supports a total of 272 interrupts and exceptions. Each has its own handler defined in a vector table. The vector table can be moved at runtime as part of context switching by an operating system if needed.

### Rich RISC-like Instruction Set

A rich set of instructions is provided, including a complete set of branches, arithmetic operations on 8, 16 and 32 bit data, with some support for 64 bit data. Several instructions map closely onto 'C' and C++ language constructs – such as conditional set, case branch and efficient function call and return instructions – thereby providing very high code density and fast execution.

A number of instructions perform specific operations needed in digital signal processing or communications systems.

### 16-bit and 32-bit Instructions

The XAP3 instruction set is completely defined by a set of 182 32-bit instructions. All instructions commonly generated by the compiler are also available as 16-bit instructions, giving very high code density. Programs can mix 32-bit and 16-bit instructions at will. No switching between modes is necessary and the processor executes all instructions at full speed. The 16-bit instructions can access all of the processor resources available to the 32-bit instructions.

### Hardware Acceleration

The processor includes a single-cycle multiplier and barrel shifter. Instruction execution is pipelined, resulting in operating speeds of up to 250MHz on a 0.13µm ASIC process.

### Targeted for Speed or Size

The XAP3 is available in two hardware implementations. XAP3a is targeted for low cost applications. XAP3b is targeted for computationally intensive applications. The programmer's model is unchanged between the two implementations but the speed of execution and relative timing of some instructions changes.

### Language Support

ANSI 'C' is supported through xap3ncc and xap3-gcc, a part of the GNU compiler collection. The gcc toolchain additionally provides C++ and compiled Java support.

### Development Tools

The xIDE integrated development environment provides a cross-platform development and debugging tool for XAP3 developers, using an industry-standard look and feel.

## 1.4 The XAP Family

The XAP3 is the third member of the Cambridge Consultants XAP family of embedded processors for ASICs and FPGAs. All three processors use the Cambridge Consultants SIF interface for debugging and the xIDE development environment for software development.

XAP1 is targeted at extremely small or extremely low power applications. It is a 16-bit Harvard architecture processor and fits in approximately 3000 gates.

XAP2 is an evolutionary development from XAP1. It is again a 16-bit Harvard architecture processor but has a larger address space and better optimisation for 'C' language applications. It requires approximately 12000 gates.

As a successor to the very successful XAP1 and XAP2, the XAP3 is a completely new development which maintains the strategy of low power, low cost and low risk, whilst providing the size and sophistication to run complex applications.

## 1.5 XAP3 Systems

The XAP3 exists as a soft IP core written entirely in Verilog. This can be targeted at either system-on-chip (ASIC) or FPGA implementations.

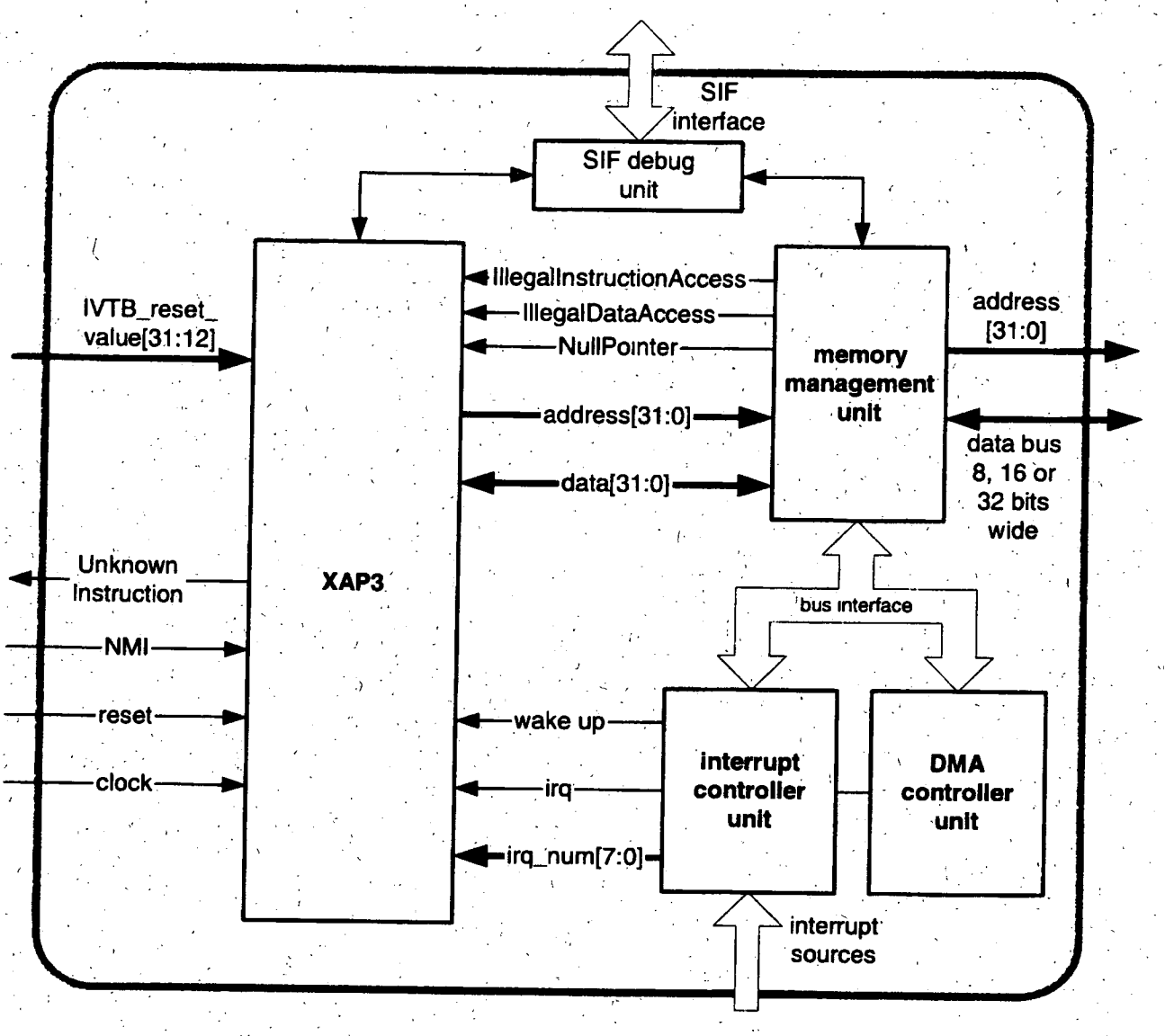
The xIDE toolset includes an instruction set simulator for XAP3. This can be extended to include models of other parts of the system, including memories and interrupts.

The XAP3 normally exists with other standard components in a system. These are:

- A Memory Management Unit (MMU) to interface with memories, typically flash and RAM.
- An Interrupt Controller Unit (ICU) to prioritise interrupt sources and provide an interrupt number to the XAP3.
- A DMA Controller Unit (DCU) to perform transfers between areas of memory without processor intervention.
- A SIF Debug Unit (SDU) to allow the system to be debugged with the xIDE integrated development environment.

Application-specific circuitry is needed for:

- System control - clock and reset generation, watchdog functions.
- Application-specific memories, peripherals and interrupt sources




## 1.6 Other Documentation

The following documents describe the software tools available for XAP3:

| Document | Contains |
|---|---|
| XAP3 Quickstart Guide<br>C7245-UM-001 | A short tour of the xIDE development environment for XAP3, intended to get users up and running quickly. |
| XAP3 GNU Binutils (Assembler, Linker and Utilities) Manual<br>C7245-UM-003 | Describes the XAP3 assembler, linker and other binary utilities. |
| XAP3 Norcroft C | Describes the xap3ncc compiler and its options, and its runtime |

| | |
|---|---|
| Compiler Manual<br>C7245-UM-004 | library |
| XAP3 GNU C/C++ Compiler Manual<br>C7245-UM-005 | Describes the xap3-gcc compiler and its options, and its runtime library. |

For the xIDE integrated development environment, the following documents are available:

| Document | Contains |
|---|---|
| xIDE User Manual<br>C7066-TM-001 | Describes the general features of xIDE. |
| xIDE Python Object Model<br>C7066-TM-003 | Describes the python object model used within xIDE. This is essential for developers wanting to use the scripting features if xIDE. This allows developers to perform automated testing or to model other parts of a system during simulation. |
| xIDE for XAP3<br>C7066-TM-006 | Describes specific features of the XAP3 plugin for xIDE. |
| XAP3 Instruction Set Quick Reference Card<br>C7245-UM-005 | A concise summary of the XAP3 instruction set |

For details of the XAP3 hardware, refer to these documents:

| Document | Contains |
|---|---|
| XAP3 Memory Management Unit Reference Manual<br>C7245-UM-006 | How to set up the XAP3 MMU for various memory configurations, including generation of chip selects and the memory exceptions. |
| XAP3 Interrupt Controller Unit Reference Manual<br>C7245-UM-007 | Description of the operation of the XAP3 interrupt controller and details of its configuration registers. |
| XAP3 DMA Controller Unit Reference Manual<br>C7245-UM-008 | Description of the operation of the XAP3 DMA controller and details of its configuration registers. |
| XAP3 Bus Interface Technical Reference<br>C7245-UM-009 | Details of the XAP3's external busses, including timing diagrams. |
| XAP3 Breakpoint and Debug Reference Manual<br>C7245-UM-010 | Details of the breakpoint registers and the SIF interface. This is of most interest to engineers producing XAP3 development tools. |
| XAP3a Hardware Reference Manual | Information needed by digital designers including the XAP3a in an FPGA or ASIC |

| | |
|---|---|
| C7245-UM-011 | |
| XAP3b Hardware Reference Manual<br>C7245-UM-012 | Information needed by digital designers including the XAP3b in an FPGA or ASIC |
| XAP3 Xilinx Quickstart Guide<br>C7245-UM-013 | How to get the XAP3 running in a Xilinx in 10 minutes. Assumes using ISE and the xEMU development board from Cambridge Consultants. |
| XAP3 Datasheet ASICs-SB-010 | Short overview of the XAP3 architecture. |

## 1.7 For Further Help

The XAP3 documentation set provides answers to most questions.

If a problem remains unsolved, contact Cambridge Consultants technical support at:

xap@cambridgeconsultants.com

## 1.8 Glossary and Acronyms

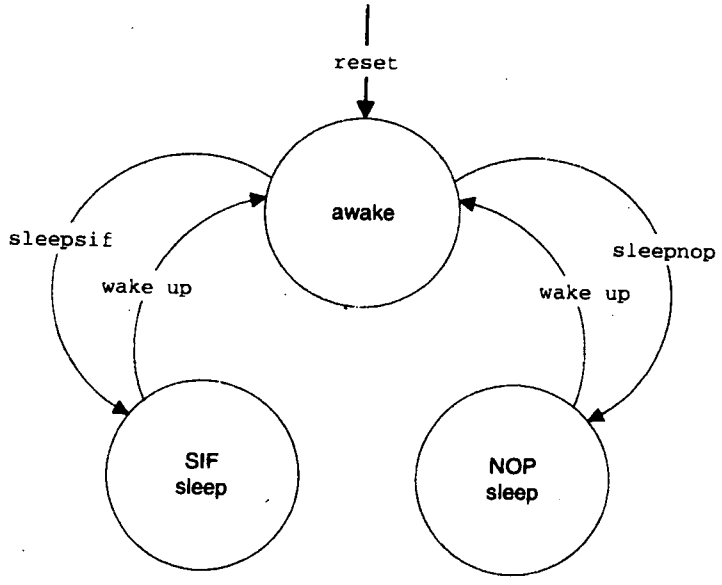
| Term | Meaning |
|---|---|
| Double-Word | An 8-byte (or 64-bit) quantity. |
| Word | A 4-byte (or 32-bit) quantity. The XAP3 is a 32-bit processor. |
| Half-Word | A 2-byte (or 16-bit) quantity. |
| Byte | An 8-bit quantity. |
| Little Endian | A system in which the least significant byte of a word is stored at the lowest memory address. The XAP3 is a little endian processor. |
| Big Endian | A system in which the most significant byte of a word is stored at the lowest memory address. |
| ALU | Arithmetic and Logic Unit. |
| DCU | DMA Controller Unit |
| ICU | Interrupt Controller Unit. |
| MMU | Memory Management Unit. |
| SDU | SIF Debug Unit. |

# 2 Programmer's Model

## 2.1 Processor States

The XAP3 has three processor states:

| Processor State | Description |
|---|---|
| Awake | The XAP3 is able to execute instructions. This is the XAP3 processor state following a reset or after a wake up from one of the sleep states. |
| SIF Sleep | The XAP3 is asleep but SIF cycles can still occur. This state is entered with the `sleepsif` instruction. |
| NOP Sleep | The XAP3 is asleep and SIF cycles are not permitted. This state is entered with the `sleepnop` instruction. |




Power consumption is significantly reduced in the two sleep states.

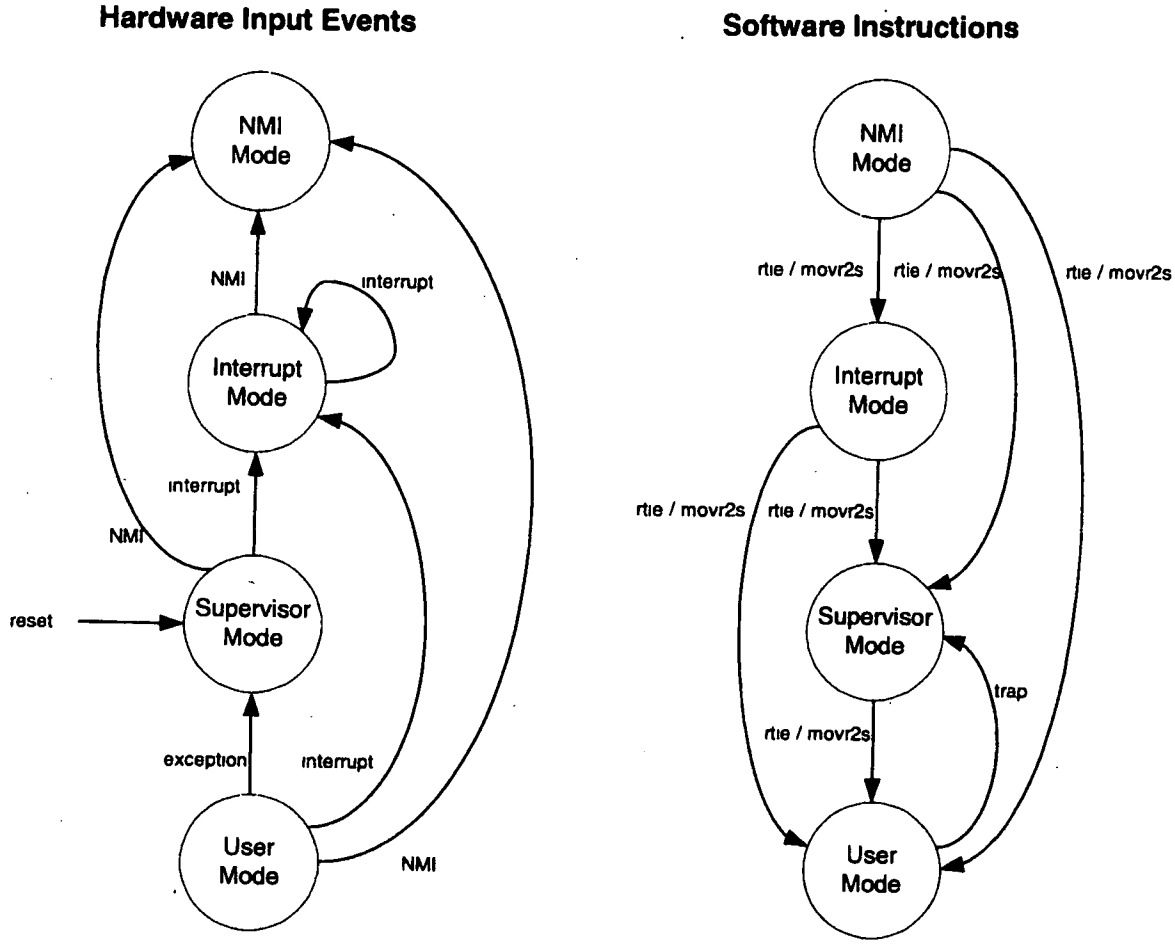
## 2.2 Processor Modes

The XAP3 has four processor modes:

| Processor Mode | | Description |
|---|---|---|
| User Mode | | Code running in User Mode cannot affect the operation of code running in the other modes. Some instructions are not permitted in User Mode. Conversely, all code in User Mode can be controlled from Supervisor Mode.<br>User Mode is suitable for untrusted application code. |
| Privileged Modes | Supervisor Mode | Code running in Supervisor Mode can execute all the XAP3 instructions and can therefore control the activity of code running in User Mode.<br>The XAP3 enters Supervisor Mode after a reset or on execution of a trap instruction.<br>Supervisor Mode is suitable for operating systems. |
| | Interrupt Mode | Interrupt Mode is entered on a hardware interrupt. |
| | NMI Mode | NMI Mode is entered on a non-maskable interrupt. |

The Supervisor, Interrupt and NMI Modes are the privileged modes.

The diagram below illustrates how mode changes take place:




It is possible to change from NMI, Interrupt and Supervisor Mode to any of the four modes by using the `movr2s` instruction. The diagram shows only the mode changes expected in normal interrupt and exception handlers.

### 2.2.1 Capabilities of the processor modes

Code running in User Mode can only access the User Mode registers and cannot write to the FLAGS register. Code running in any of the privileged modes can access its own registers and the User Mode registers.

Code running in Supervisor Mode cannot access Interrupt or NMI Mode registers, and vice versa. In order to initialise Interrupt and NMI Mode registers, it is necessary to disable interrupts and perform mode changes by writing to the FLAGS register with the `movr2s` instruction.

| | | current processor mode | | | |
|---|---|---|---|---|---|
| | | **User** | **Supervisor** | **Interrupt** | **NMI** |
| **can access these** | **User** | yes | yes | yes | yes |
| | **Supervisor** | no | yes | no | no |

| | | | | | |
|---|---|---|---|---|---|
| these registers | Interrupt | no | no | yes | yes (Interrupt Mode R2 and R15 only) |
| | NMI | no | no | no | yes |

### 2.2.2 Typical usage of processor modes

#### *Simple applications*

Simple applications may execute in Supervisor Mode and make no use of User Mode. Interrupt Mode and NMI Mode are used for interrupt and NMI operations.

#### *Complex applications*

For complex applications it is likely that most of the processing will occur in User Mode. Services provided by an operating system will execute in Supervisor Mode. User Mode code can call Supervisor Mode code using the `trap` instructions.

## 2.3 Registers

The XAP3 register set includes:

- 16 Normal registers called R0 to R15. Four of the normal registers are shadowed in hardware to assist with processor mode switches.
- The Program Counter, called PC.
- 7 Special registers. These are FLAGS, SFLAGS_S, SFLAGS_I, SFLAGS_N, GP_U, GP_S, GP_I, IVTB and BRKE.
- 16 Breakpoint registers, all starting with the prefix BRK.

In the descriptions of the registers which follow, hardware registers are in upper case (for example, R1) whilst the assembler name for a register is in lower case (for example, `%r1`).

### 2.3.1 Normal registers

The XAP3 has sixteen normal registers, called R0 to R15. Registers R1, R2, R14 and R15 are shadowed in hardware such that the hardware register used depends on the current processor mode.

| Normal Registers | User Mode Registers 31 0 | Supervisor Mode Registers 31 0 | Interrupt Mode Registers 31 0 | NMI Mode Registers 31 0 | Assembler Syntax |
|---|---|---|---|---|---|
| Constant Zero | R0 | R0 | R0 | R0 | %r0 |
| Return Value and function Argument 1 | R1_U | R1_S | R1_I | R1_N | %r1 |
| Function Argument 2 | R2_U | R2_S | R2_I | R2_N | %r2 |
| Function Argument 3 | R3 | R3 | R3 | R3 | %r3 |
| Function Argument 4 | R4 | R4 | R4 | R4 | %r4 |
| Function Argument 5 | R5 | R5 | R5 | R5 | %r5 |
| | R6 | R6 | R6 | R6 | %r6 |
| | R7 | R7 | R7 | R7 | %r7 |
| Accumulator 0 { | R8 | R8 | R8 | R8 | %r8 |
| | R9 | R9 | R9 | R9 | %r9 |
| Accumulator 1 { | R10 | R10 | R10 | R10 | %r10 |
| | R11 | R11 | R11 | R11 | %r11 |
| | R12 | R12 | R12 | R12 | %r12 |
| | R13 | R13 | R13 | R13 | %r13 |
| Link Register (LR) | R14_U | R14_S | R14_I | R14_N | %r14 |
| Stack Pointer (SP) | R15_U | R15_S | R15_I | R15_N | %r15 |

The diagram shows which registers are used in each of the modes. The privileged modes are able to access R1_I, R2_U, R14_U and R15_U by using the `ldm.u` instruction.

Some of the normal registers are allocated dedicated functions in the XAP3 architecture:

| Assembler syntax | Hardware register name | Notes |
|---|---|---|
| `%r0` | R0 | R0 is a constant zero register.<br>R0 can be used as a destination register. In this situation the write does not take place but the flags are updated as if it had done. |
| `%r1` | R1_U or R1_S or R1_I or R1_N | The hardware register used as `%r1` depends on the processor mode. |
| `%r2` | R2_U or R2_S or R2_I | The hardware register used as `%r2` depends on the processor mode. R2_I is used for Interrupt and NMI Modes. |
| `%r3` | R3 | |
| `%r4` | R4 | |
| `%r5` | R5 | |
| `%r6` | R6 | |
| `%r7` | R7 | |
| `%r8` | R8 | Accumulator 0 bits [31:0] |
| `%r9` | R9 | Accumulator 0 bits [63:32] |
| `%r10` | R10 | Accumulator 1 bits [31:0] |

| | | |
|---|---|---|
| %r11 | R11 | Accumulator 1 bits [63:32] |
| %r12 | R12 | |
| %r13 | R13 | |
| %r14 | R14_U or R14_S or R14_I or R14_N | The hardware register used as %r14 depends on the processor mode.<br>%r14 is used as a Link Register during subroutine calls. |
| %r15 | R15_U or R15_S or R15_I | The hardware register used as %r15 depends on the processor mode. R15_I is used for Interrupt and NMI Modes.<br>%r15 is used as a Stack Pointer. |

### *64-bit accumulators*

Registers R8 and R9, and R10 and R11 can be combined to create two 64-bit accumulators. Each 64-bit accumulator is referred to by the name of the lower register:

| 64-bit accumulator name | | 63 32 31 0 |
|---|---|---|
| | %r8 | R9 R8 |
| | %r10 | R11 R10 |

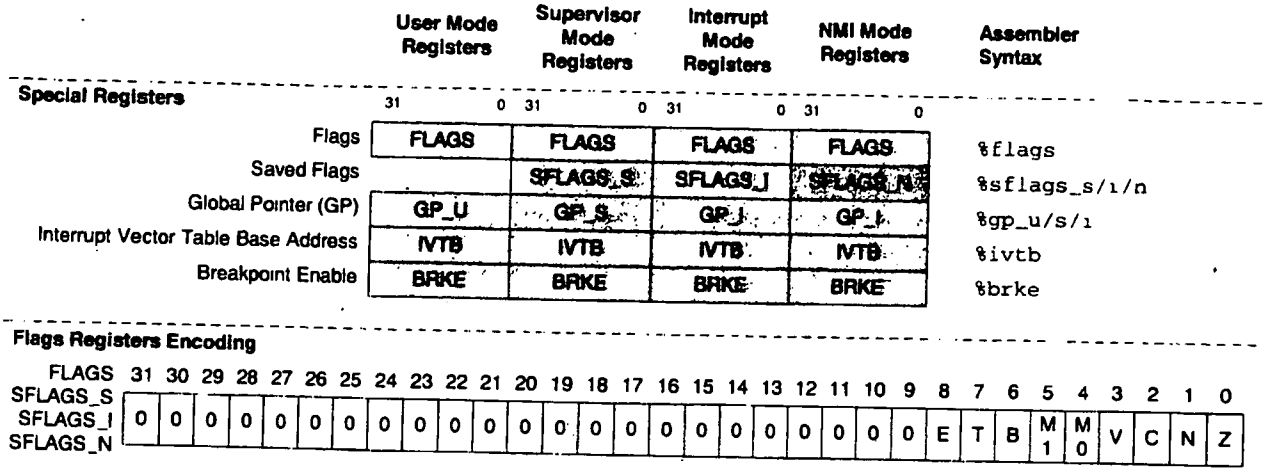
The 64-bit accumulators are used in the `acc.64.r`, `shiftl.64.r`, `shiftr.64s.i` and `shiftr.64s.i` instructions.

## 2.3.2 Program Counter

The program counter is a 32-bit register that contains the address of the next instruction to be executed. The program counter cannot be accessed directly as a register but is implictly used by all the .p instructions, the bsr instructions and the `rtie` instruction.

The XAP3 forces the least-significant bit of the program counter to be zero. Attempts to set the least significant bit throw an `IllegalCodeAddress` exception.

### 2.3.3 Special registers




The `movr2s` and `movs2r` instructions allow the privileged modes access to the all the special registers. The special registers are described in the following table.

| Assembler syntax | Hardware register Name | Notes |
|---|---|---|
| `%flags` | FLAGS | The flags register. This is used in all processor modes. |
| `%sflags_s` | SFLAGS_S | Saved flags. The processor copies FLAGS into SFLAGS_S when an exception occurs, and restores FLAGS from SFLAGS_S when an `rtie` instruction is executed in Supervisor Mode. |
| `%sflags_i` | SFLAGS_I | Saved flags. The processor copies FLAGS into SFLAGS_I when an interrupt occurs, and restores FLAGS from SFLAGS_I when an `rtie` instruction is executed in Interrupt Mode. |
| `%sflags_n` | SFLAGS_N | Saved flags. The processor copies FLAGS into SFLAGS_N when an NMI occurs, and restores FLAGS from SFLAGS_N when an `rtie` instruction is executed in NMI Mode. |
| `%gp_u` | GP_U | The Global Pointer used by `mov.g` instructions in User Mode. |
| `%gp_s` | GP_S | The Global Pointer used by `mov.g` instructions in Supervisor Mode. |
| `%gp_i` | GP_I | The Global Pointer used by `mov.g` instructions in Interrupt and NMI Modes. |
| `%ivtb` | IVTB | The Interrupt Vector Table base address. |
| `%brke` | BRKE | The breakpoint enable register. |

Each register is described in the following sections.

### Flags register

The table below describes the format of the FLAGS, SFLAGS_S, SFLAGS_I and SFLAGS_N registers:

| Bit | Flag | Name | Description |
|---|---|---|---|
| 0 | Z | Zero | Updated by most instructions according to the result of their operation. Refer to the individual instruction descriptions in the Instruction Set Reference for details of the flag behaviour |
| 1 | N | Negative | |
| 2 | C | Carry | |
| 3 | V | Overflow | |
| 5:4 | M[1.0] | Mode | Indicates the current processor operating mode:<br>00 Supervisor Mode<br>01 User Mode<br>10 Interrupt Mode<br>11 NMI Mode |
| 6 | B | Break | Controls the processor's response to a `brk` instruction or a hardware breakpoint event. Refer to section "Break" for further details. |
| 7 | T | Single Step | If set, each User Mode instruction that is executed throws the `SingleStep` exception. |
| 8 | E | Interrupt Enable | If set, interrupts are enabled. Non-maskable interrupts (NMI) cannot be disabled.<br>The E bit is cleared when an interrupt or exception occurs. |
| 31:9 | unused | | Read as zero. |

The following events cause the flags register to be updated:

- A reset.
- An instruction which updates the flags.
- Code running in one of the privileged modes executes a `movr2s` instruction.
- Code running in one of the privileged modes executes an `irqd` or `irqe` instruction.
- An exception.
- An interrupt or NMI.

### Saved Flags registers

The Saved Flags registers (SFLAGS_S, SFLAGS_I, SFLAGS_N) are used to save the contents of the FLAGS register when the processor changes mode. An `rtie` instruction copies one of the saved flags registers into FLAGS to restore the processor states.

To support nested interrupts, the SFLAGS_I register should be saved on the stack.

### *Global Pointer register*

The GP registers supply a base address for a program's data segment. Memory reads and writes are made relative to the active GP register.

### *Interrupt Vector Table Base register*

The IVTB register sets the base address of the IVT

The initial value of IVTB at reset is controlled by the IVTB_reset_value[31:12] signals supplied to XAP3. This allows the IVT to be located to any 4KByte boundary after reset.

The program is able to change IVTB to point to any word boundary.

### *Break Enable register*

The BRKE register is used in conjunction with the sixteen breakpoint registers. Details are available in document "XAP3 Breakpoint and Debug Reference Manual" from Cambridge Consultants.

## 2.3.4 Breakpoint registers

The XAP3 has sixteen breakpoint registers and one breakpoint enable register. Details of these are available in document "XAP3 Breakpoint and Debug Reference Manual" from Cambridge Consultants.

The breakpoint registers are accessed with the `movr2b` and `movb2r` instructions and are accessible from the privileged modes only.

| | User Mode Registers | Supervisor Mode Registers | Interrupt Mode Registers | NMI Mode Registers | Assembler Syntax |
|---|---|---|---|---|---|
| **Breakpoint Registers** | | 31 0 | 31 0 | 31 0 | |
| Breakpoint 0 | BRK0 | BRK0 | BRK0 | BRK0 | %brk0 |
| Breakpoint 1 | BRK1 | BRK1 | BRK1 | BRK1 | %brk1 |
| Breakpoint 2 | BRK2 | BRK2 | BRK2 | BRK2 | %brk2 |
| Breakpoint 3 | BRK3 | BRK3 | BRK3 | BRK3 | %brk3 |
| Breakpoint 4 | BRK4 | BRK4 | BRK4 | BRK4 | %brk4 |
| Breakpoint 5 | BRK5 | BRK5 | BRK5 | BRK5 | %brk5 |
| Breakpoint 6 | BRK6 | BRK6 | BRK6 | BRK6 | %brk6 |
| Breakpoint 7 | BRK7 | BRK7 | BRK7 | BRK7 | %brk7 |
| Breakpoint 4 Count | BRK4COUNT | BRK4COUNT | BRK4COUNT | BRK4COUNT | %brk4count |
| Breakpoint 5 Count | BRK5COUNT | BRK5COUNT | BRK5COUNT | BRK5COUNT | %brk5count |
| Breakpoint 6 Count | BRK6COUNT | BRK6COUNT | BRK6COUNT | BRK6COUNT | %brk6count |
| Breakpoint 7 Count | BRK7COUNT | BRK7COUNT | BRK7COUNT | BRK7COUNT | %brk7count |
| Breakpoint 6 Mask | BRK6MASK | BRK6MASK | BRK6MASK | BRK6MASK | %brk6mask |
| Breakpoint 7 Mask | BRK7MASK | BRK7MASK | BRK7MASK | BRK7MASK | %brk7mask |
| Breakpoint 6 Data | BRK6DATA | BRK6DATA | BRK6DATA | BRK6DATA | %brk6data |
| Breakpoint 7 Data | BRK7DATA | BRK7DATA | BRK7DATA | BRK7DATA | %brk7data |

**BRKE Register Encoding**

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BRKE | 0 | E7 | R7 | W7 | 0 | E6 | R6 | W6 | 0 | E5 | R5 | W5 | 0 | E4 | R4 | W4 | 0 | E3 | R3 | W3 | 0 | E2 | R2 | W2 | 0 | E1 | R1 | W1 | 0 | E0 | R0 | W0 |

## 2.4 Pipeline

The XAP3 is a pipelined processor. XAP3a has a two-stage pipeline. XAP3b has a four stage pipeline. For further details, including details of instruction cycle counts, refer to the appropriate Hardware Reference Manual.

## 2.5 Memory Architecture

The XAP3 can address 4GBytes ($2^{32}$ bytes) of memory. The memory space is linear and byte addressable and contains both code and data.

32-bit and 16-bit instructions are aligned to a half-word (2 bytes) boundary.

For optimum speed, data objects are aligned to byte, half-word or word boundaries, depending on their type. A compiler option allows data objects to be densely packed. The XAP3 accesses such unaligned data objects natively and with no programmer effort.

The stacks and interrupt vector table are aligned to a word boundary (4 bytes).

## 2.6 Stack Operation

The XAP3 stacks start at high addresses and grow downwards in memory. The stacks are 32 bits wide. Push operations decrease the stack pointer by a multiple of four bytes. Pop operations increase the stack pointer by a multiple of four bytes.

Register R15 is used as the stack pointer and points to the last used location on the stack. Separate stacks exist for User Mode, Supervisor Mode and Interrupt Mode.

The compilers operate a fully covered stack. When accessing stack data (local automatic variables or parameters), stack offsets are always positive or zero.

## 2.7 Reset

Following a reset, the XAP3 state is:

- Processor awake.
- Processor Mode is Supervisor.
- All registers are zero.
- All special registers are zero.
- All breakpoint registers are zero.
- IVTB loaded from external pins to set the base address of the IVT.
- Program counter is set to the IVTB reset value.

The XAP3 fetches the first instruction from the IVT base address.

With the flags register being zero, interrupts are disabled, single stepping is disabled and the break exception is disabled.

## 2.8 Interrupts and Exceptions

Exceptions are generated by code running in User Mode and are generally the result of a `trap` instruction or a problem with the code. In response to the exception, the XAP3 moves to Supervisor Mode and disables interrupts. Once in Supervisor Mode, no further exceptions occur.

Interrupts can occur in any mode whilst interrupts are enabled in the FLAGS register. The XAP3 moves to Interrupt Mode and disables interrupts. The XAP3 receives an interrupt number (typically from an external interrupt controller) and branches to an appropriate interrupt handler. At the end of the interrupt processing, the XAP3 returns to the interrupted code with the `rtie` instruction.

Non-maskable interrupts cannot be disabled and can occur in any mode. The XAP3 moves to NMI Mode. At the end of the processing for the non-maskable interrupt, the XAP3 can return to the interrupted code with the `rtie` instruction

Interrupts which occur whilst the XAP3 is in NMI Mode are remembered and are actioned after the exit from NMI Mode with the `rtie` instruction.

### 2.8.1 Exception processing in User Mode

When an exception occurs, the XAP3 sets up the Supervisor Mode registers, disables interrupts, and then branches to the appropriate exception handler. The information in the Supervisor Mode registers on entry to the handler allows the handler to deal with the exception condition in the User Mode code. The Supervisor code may choose to allow the User Mode code to continue, or it may take action to kill and restart the User Mode code.

In detail, the XAP3 exception processing is:

- Store the current flags in SFLAGS_S.
- Store the current PC in LR_S. This is the address of the instruction following the instruction which threw the exception.
- Set up R1_S. If the exception was caused by a trap instruction, R1_S is loaded with the argument from the trap instruction. Otherwise, R1 is loaded with the address of the instruction which caused the exception.
- The XAP3 is put into Supervisor Mode.
- Interrupts are disabled.
- The XAP3 branches to an address in the Interrupt Vector Table. The Interrupt Vector Table would normally contain a branch to an exception handler.

Returning from an exception is done with the `rtie` instruction. If LR_S is not changed by the exception handler, executing an `rtie` will cause the User Mode code to resume at the instruction following instruction which threw the exception.

Interrupts are disabled on entry to Supervisor Mode. The exception handler can re-enable them if required with the `irqe` instruction.

### 2.8.2 Exception processing in the privileged modes

Exceptions do not occur in the privileged modes. Exception-generating conditions are treated as follows:

| Exception Condition | Behaviour in the privileged modes |
|---|---|
| `Trap` | A `trap` instruction is treated as a `nop`. |
| `SingleStep` | Single stepping is not possible in the privileged modes, so this exception cannot occur. |
| `Break` | A break condition (either a `brk` instruction or hardware breakpoint) is treated as a halt or a nop, depending on whether debug mode is |

| | |
|---|---|
| | enabled or not. Refer to section "Break" for more details. |
| UnknownInstruction | The XAP3 sets an UnknownInstruction output. The system designer can route this signal into the system's interrupt, NMI or reset circuitry. |
| DivideByZero | The destination register is not updated.<br>The Z and N flags are updated according to the contents of the destination register.<br>The V flag is set if the operation was signed. The C flag is set if the operation was unsigned. |
| NullPointer | Null pointer dereferences are allowed in the privileged modes. |
| IllegalInstruction | All instructions are legal in the privileged modes, so this exception cannot happen. |
| IllegalInstructionAccess and IllegalDataAccess | These conditions are ignored. This allows an operating system to access memory belonging to all tasks. |
| IllegalCodeAddress | The least significant bit of the offending address is ignored and the instruction is executed as if the least significant bit was zero. |
| IllegalStackAddress | The least significant two bits of the stack pointer are ignored and the stack operation is executed as if the two least significant bits were zero. |

### 2.8.3 Interrupt processing

The XAP3 has a single interrupt input (IRQ) and a single NMI input. If interrupts are enabled, activating the IRQ input causes the XAP3 to switch to Interrupt Mode when the current instruction finishes. The XAP3 then receives an interrupt number which is used to calculate an index into the Interrupt Vector Table. Entries in the Interrupt Vector Table will typically be branches to interrupt handlers.

Non-Maskable Interrupts can occur at any time and cannot be disabled. Activating the NMU input causes the XAP3 to switch to NMI Mode when the current instruction finishes. The additional shadow registers in NMI Mode ensure that an NMI handler can restore the processor to its state immediately prior to the NMI event.

In detail, the XAP3 interrupt processing is:

- Store the current flags in SFLAGS_I or SFLAGS_N.
- Store the current PC in LR_I or LR_N. This is the address of the first instruction to be executed on return from the interrupt handler.
- The XAP3 is put into Interrupt or NMI Mode.
- Interrupts are disabled.
- The XAP3 branches to an address in the Interrupt Vector Table. The Interrupt Vector Table would normally contain a branch to an interrupt handler.

### NMI Notes

The NMI interrupt cannot be disabled. Multiple NMIs may cause the stack to grow if an NMI handler is itself interrupted by another NMI. As NMI Mode and Interrupt Mode share the same stack (R15_I), the Interrupt Mode stack should have sufficient space to allow for an NMI and its stack frame.

NMI Mode shares R2 and R15 with Interrupt Mode. Care is necessary when using these registers in an NMI exception handler.

## 2.8.4 Returning from interrupts and exceptions

The `rtie` instruction is used to return from interrupts and exceptions and is implemented as:

```
if (FLAGS.Mode == NMIMode) then
        FLAGS = SFLAGS_N
        PC    = LR_N
elseif (FLAGS.Mode == InterruptMode) then
        FLAGS = SFLAGS_I
        PC    = LR_I
elseif (FLAGS.Mode == SupervisorMode) then
        FLAGS = SFLAGS_S
        PC    = LR_S
endif
```

Thus the null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code.

## 2.8.5 Nested interrupts

Interrupts are disabled on entry to Interrupt Mode. To support nestable interrupts some additional work is required of the interrupt handler to save the processor state prior to re-enabling interrupts with the `irqe` instruction.

The interrupt handler must similarly restore the processor state prior to returning.

Example code for nested interrupts is given in section "Nested interrupt prologue and epilogue".

## 2.8.6 Interrupt vector table

The IVT sits in memory with its base defined by the `IVTB` register. At reset, `IVTB` is loaded from external pins to the XAP3, which locates the IVT to any 4KByte boundary. By changing `IVTB`, the IVT can be moved during a program's execution. The IVT can be located in RAM or ROM. The start address of the IVT must be 32-bit aligned.

Each entry in the IVT is one 32-bit instruction. These will often be either:

- `bra.p` to the start of the relevant handler, or
- the null handler (the `rtie` instruction).

The IVT manages both exceptions and interrupts.

The layout of the IVT is shown below. The reserved exceptions are for future expansion.

| Offset into IVT | Exception or Interrupt | Resulting XAP3 mode |
|---|---|---|
| 0x0000 | Reset | Supervisor |
| 0x0004 | Non-Maskable Interrupt | NMI |
| 0x0008 | `Trap` Exception | Supervisor |
| 0x000C | `SingleStep` Exception | Supervisor |
| 0x0010 | `Break` Exception | Supervisor |
| 0x0014 | `UnknownInstruction` Exception | Supervisor |
| 0x0018 | Reserved | Supervisor |
| 0x001C | Reserved | Supervisor |
| 0x0020 | `DivideByZero` Exception | Supervisor |
| 0x0024 | `NullPointer` Exception | Supervisor |
| 0x0028 | `IllegalInstruction` Exception | Supervisor |
| 0x002C | `IllegalInstructionAccess` Exception | Supervisor |
| 0x0030 | `IllegalDataAccess` Exception | Supervisor |
| 0x0034 | `IllegalCodeAddress` Exception | Supervisor |
| 0x0038 | Reserved | Supervisor |
| 0x003C | Reserved | Supervisor |
| 0x0040 | Interrupt, interrupt number 0 | Interrupt |
| 0x0044 | Interrupt, interrupt number 1 | Interrupt |
| 0x0048 | Interrupt, interrupt number 2 | Interrupt |
| ... | .. | |
| 0x0424 | Interrupt, interrupt number 255 | Interrupt |

## 2.8.7 Exceptions

### *Trap*

The two trap instructions cause the XAP3 to move into Supervisor Mode and may be used for User Mode code to call an operating system. The argument to the trap instruction (either a register or an immediate) is passed to the trap exception handler in R1_S.

### *SingleStep*

When the T bit is set in the `FLAGS` register, this exception is triggered after every User Mode instruction, but only if that instruction has not caused any other exception (e.g., `DivideByZero`). This supports run-mode debugging and single stepping of User Mode applications.

### Break

The Break exception can be caused by either a `brk` instruction or by a break condition being satisfied in the breakpoint registers.

Break events are handled according to this logic.

```
if ( (FLAGS.Mode == User) && (FLAGS.B == 1) ) then
    throw break exception
else if debug enabled then
    halt
else
    nop
endif
```

Debug mode is enabled using the SIF.

When runing a program in the xIDE simulator or on an emulator, debug mode is always enabled. Any break conditions will halt the processor unless the processor is in User Mode and the B flag is set.

The `brk` instructions are valid in User Mode. An operating system can prevent a User Mode task from stopping the processor by setting the B bit of the `FLAGS` register.

### UnknownInstruction

All opcodes that do not correspond to a valid instruction trigger this exception.

On entry to the handler, R1 points to the undefined instruction just executed

### DivideByZero

All forms of divide and remainder instructions check the value of the denominator. If it is zero, then this throws a `DivideByZero` exception.

The following instructions can cause this exception:

- `div.u.r`
- `div.s.r`
- `rem.u.r`
- `rem.s.r`

### NullPointer

The XAP3 MMU detects attempts by a program to access memory a range of addresses starting from zero. For further details, refer to the XAP3 MMU Reference Manual. Such accesses are normally the result of a program using an un-initialised pointer variable.

LR points to the next instruction to be executed and R1 points to the instruction just executed.

The following instructions can cause this exception:

- `ld.i, ld.16s.i, ld.16z.i, ld.8.s.i, ld.8.z.i`
- `ld.r, ld.16s.r, ld.16z.r, ld.8.s.r, ld.8.z.r`
- `ldm, ldm.x`
- `st.i, st.16.i, st.8.i`
- `st.r, st.16.r, st.8.r`
- `stm, stm.x`
- `push, push.r`
- `pop, pop.ret`

`ldm.u` and `stm.u` do not generate this exception, because they are not valid instructions in User Mode.

### *IllegalInstruction*

In an application using an operating system or supervisor, the XAP3 executes most code in User Mode. This mode is restricted in its access rights to certain instructions; executing any privileged instruction in User Mode will trigger an `IllegalInstruction` exception.

The following instructions cause this exception, because they provide access to the `FLAGS` register and could be used to create mode changes:

- `rtie`
- `ldm.u`
- `stm.u`
- `movr2s, movs2r`
- `movr2b, movb2r`
- `irqe, irqd`

The following instructions cause this exception, because they alter the behaviour of the XAP3 core.

- `sleepsif, sleepnop`
- `halt`
- `sif`

The `sif` instruction is not allowed in User Mode, because it can alter the timing of accesses from external devices to the registers and memory of the processor. If SIF cycles are required

in User Mode code, the SIF instruction should be wrapped as a trap handler and made available via a `trap` instruction.

### *IllegalInstructionAccess and IllegalDataAccess*

These exceptions are triggered by the external MMU in response to the XAP3 core accessing memory that violates the current access rights (e.g., reading from memory that does not belong to the currently executing process, or writing to memory that is tagged as read-only). For further details, refer to the XAP3 MMU Reference Manual.

`IllegalInstructionAccess` is thrown when an instruction fetch violates the access rights. `IllegalDataAccess` is thrown when an instruction's data read or write violates the access rights.

On entry to the handler R14 points to the next instruction to be executed and R1 points to the instruction that caused the exception. The MMU places the address that caused the exception in a memory mapped register accessible to the exception handler code.

These exceptions can be used to implement virtual memory systems or to implement memory protection schemes between different processes.

`IllegalDataAccess` can be caused by the following instructions:

- `ld.i, ld.16s.i, ld.16z.i, ld.8.s.i, ld.8.z.i`
- `ld.r, ld.16s.r, ld.16z.r, ld.8.s.r, ld.8.z.r`
- `ldm, ldm.x`
- `st.i, st.16.i, st.8.i`
- `st.r, st.16.r, st.8.r`
- `stm, stm.x`
- `push, push.r`
- `pop, pop.ret`

`ldm.u` and `stm.u` do not generate an exception, because they are not valid instructions in User Mode.

### *IllegalCodeAddress*

All instructions are half-word aligned, such that `PC[0]` is zero. This exception is triggered when an odd value is written to PC, (i.e., `PC[0]` is set).

The following instructions can cause this exception.

- `bra.r`
- `bsr.r`

- `pop.ret`
- `rtie`

It is not possible for the immediate forms of the branch instructions throw this exception, because they cannot encode illegal addresses.

### *IllegalStackAddress*

An `IllegalStackAddress` exception is thrown if the stack pointer is not correctly aligned when a stack operation (push or pop) is performed. The stack pointer must always be on a word boundary.

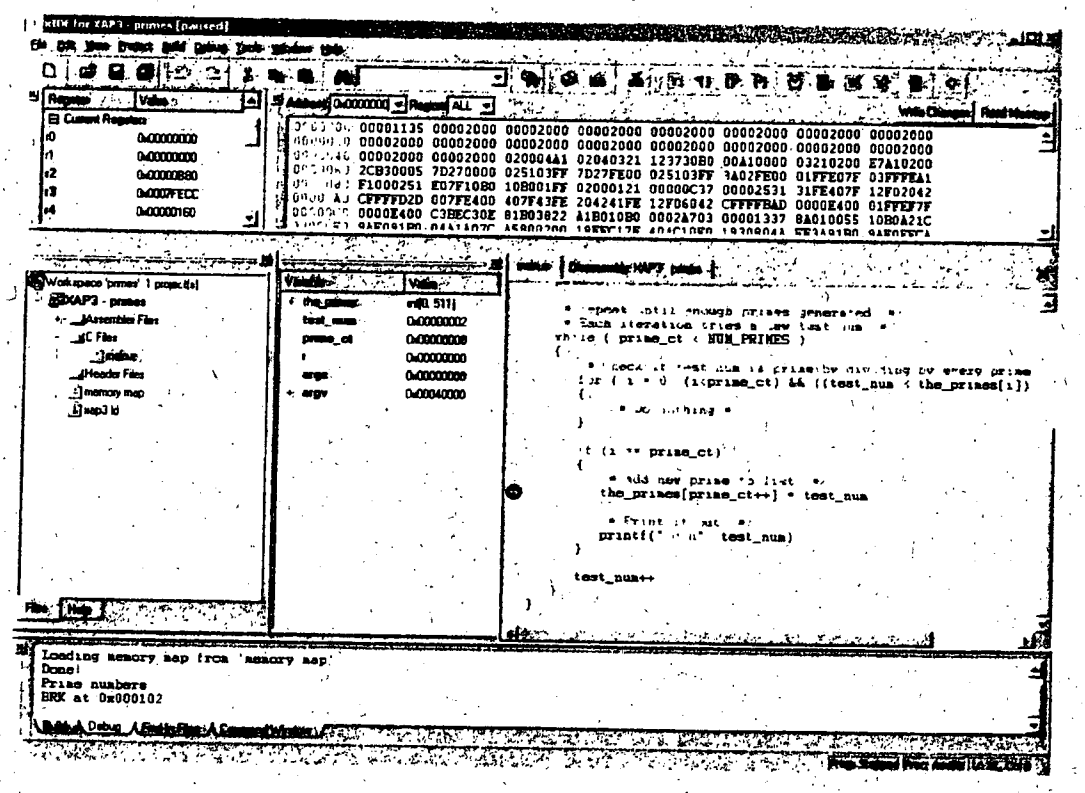
The following instructions can cause this exception:

- `push`
- `push.r`
- `pop`
- `pop.ret`

## 2.9 Debugging

### *The xIDE Integrated Development Environment*

Systems containing a XAP3 are developed and debugged with the xIDE integrated development environment using the SIF interface.



xIDE provides:

- An integrated development and unified debug tool interface.
- Multiprocessor support.
- Familiar interface with HTML-based on-line help.
- Extensible architecture through software plug-ins.
- Project navigator to project files and program builds.
- Built-in multiple-document text editor with syntax highlighting.
- Customisable docking windows, toolbars and GUI widgets for system-specific requirements.
- Integrated Python command line. Python macros automate frequently used tasks.
- Supports Python macros to automate and simulate target system functionality.
- Multiple document windows for source code, browsing memory, debug output, register views, peripherals and variables watch.
- Support for simultaneous debug of multi-processor designs.
- Support for stop-mode and run-mode debugging.

### Stop-Mode Debugging

In stop-mode debugging, all code running on the XAP3 is stopped when a breakpoint is encountered or whilst single stepping. This is the normal debugging situation.

### Run-Mode Debugging

The XAP3 instruction set supports run-mode debugging of User Mode code. An on-chip debugger running in Supervisor Mode can use the `SingleStep` and `Break` exceptions to debug code running in User Mode.

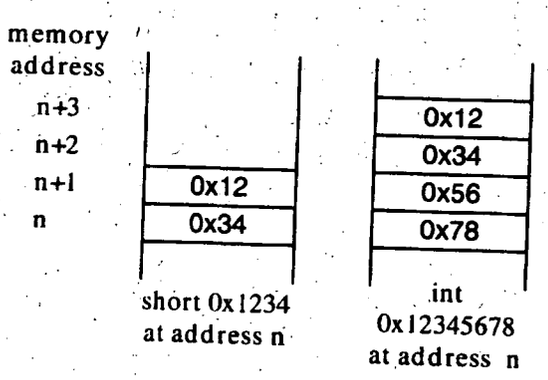
# 3 C/C++ Language Interface

## 3.1 Data Types

The XAP3 has instructions to operate on 8, 16, 32 or 64 bit wide data. The load and store instructions transfer 8, 16, or 32 bit wide data to and from memory.

### *Endianness*

Data is stored in memory in a little-endian byte order:




### *Non-native data types*

There is no hardware support for transferring 64-bit data to and from memory. However, 64-bit data should be stored in memory in a little endian format.

Similarly, there is no hardware support for floating point data types. The format used by the compilers is the IEEE 754 standard, implemented as little endian.

## 3.2 Data Alignment

The XAP3 supports accesses to 16-bit and 32-bit data at any memory address. Accesses to unaligned data are performed natively and with no programmer effort.

Alignment of data objects within memory is described in detail in the compiler manuals.

### 3.2.1 Aligned data

Aligned data produces the fastest code and is the normal model for the compilers.

Data objects are aligned to start on word bondaries. Fields within structures are similarly aligned to word boundaries, the compilers adding padding bytes as necessary to achieve this alignment.

### 3.2.2 Unaligned data

A compiler option allows structures to be packed. This prevents the compiler from adding padding bytes between fields and can result in fields lying at non-word-aligned addresses. The XAP3 supports accesses to such data natively.

Unaligned data can be accessed with the `ld.*`, `st.*`, `ldm.*`, `stm.*` and `swap.r` instructions.

Typically, two memory accesses are required for each unaligned half-word operation, and up to three for an unaligned word operation. There is therefore a runtime penalty associated with unaligned data.

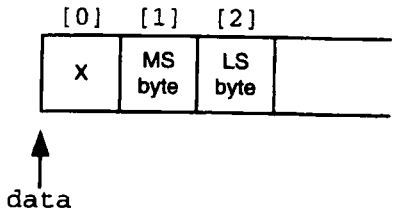
The MMU may detect accesses to unaligned data. This may provide useful debugging information, or, if speed of execution is an issue, it allows the programmer to restructure the data to remove the unaligned access. Refer to the XAP3 MMU Reference Manual for more details.

### 3.2.3 Methods of accessing unaligned data

There are three methods of accessing unaligned data. The methods vary in their speed, readability and portability.

| | **Relative duration to access an unaligned half-word** | **Readability** | **Portability** |
|---|---|---|---|
| Read the data as a byte stream | Slow | Good | Good |
| Use 'C' to produce unaligned accesses | Medium | Good | Poor |
| Use a flip instruction | Very fast | Written in assembly language or with compiler intrinsics | XAP3-specific |

The following sections illustrate the three methods extracting a single little-endian 16-bit integer from an unaligned big-endian byte stream. The incoming data is in a byte buffer called data.



### Read the data as a byte stream

The data is read as a sequence of bytes which are reassembled into the correct order. This code is entirely portable across compilers and architectures but requires two memory reads per half-word or four memory reads per word extracted.

```
extern uint8 *data;
int y = (*(data+1) << 8) | (*(data+2)); // Two byte reads
```

### Use 'C' to produce unaligned accesses

The first assignment generates an unaligned read from the data buffer. This code will not work on architectures that do not support unaligned memory accesses but is possible on XAP3.

```
extern uint8 *data;
int y = *(int *)(data+1);                    // Unaligned read
y = ((y&0xFF00)>>8) | ((y&0x00FF)<<8)        // Convert endianness
```

### Use the flip instruction

This code achieves the required result in four instructions.

```
extractAndFlip:
mov.g       %r1, #data          // Get address of data in R1
ld.16z.i    %r1, @(0, %r1)      // Word-aligned 16-bit read
flip.16b.r  %r1, %r1, %r0, 1    // Select required bytes and
                                //   convert endianness
bra.r       %r14                // Return to caller with flags
                                //   set correctly
```

Typically an incoming byte stream needs to be converted to a 'C' data structure. In most cases, the ld/flip combination requires at most one memory read per half-word or word extracted. A sequence of ld/flip instructions provides an efficient method of performing the conversion.

## 3.3 Calling Convention

The branch instructions (normally `bsr.p` and `bsr.a`) are used to make function calls. The effect of `bsr.p` or `bsr.a` is to transfer the address of the next instruction to the link register, R14, and to transfer the destination address to the PC. Control is returned to the calling function when the contents of R14 are transferred to PC. In practice, the compilers generate the `bsr.p` instruction.

The XAP3 calling convention is:

- On entry to a function, R14 (the Link Register) is the return address. The bsr instructions load R14 automatically.
- The called function does not need to preserve the contents of R1, R2, R3, R4, R5 and R14. R14 is the link register.

- The called function is responsible for preserving the remaining registers (R6 – R13, R15) across function calls. This convention is implemented in the C compiler. It is not a hardware property.
- R15 (Stack Pointer) is used to operate a fully covered stack. All used stack is at or above the stack pointer, and the stack pointer indicates the last used location.
- Arguments are passed to functions in R1, R2, R3, R4, R5 then on the stack. When a 16-bit value is passed to a function in a register, the whole register is used. The upper bits are sign or zero extended according to the type. Structures that are passed in registers are converted to a sequence of 32-bit words.
- The called function places the C function return value in R1.
- Special functions may store extra return data in R2, R3, R4, R5.
- Code is compiled independent of processor mode.
- When the processor is interrupted, some assembler code must be executed to store the contents of R3-R5 before the C routine can be called. This is not needed for R1 or R2 as they have hardware shadow registers.

The flags are set according to the result of comparing the return value, held in R1, with zero. The `pop.ret` instruction performs this compare automatically. Any function call can be followed immediately by any of the conditional branch or conditional set instructions.

For functions that have no return value (`void` functions in 'C') the flags are undefined.

## 3.4 Application Models

The XAP3 can support various application models.

### 3.4.1 Static linkage

This is the basic application model where the locations of all functions and data are known at link time.

### 3.4.2 Dynamc linkage

The XAP3 supports systems where applications can be loaded to memory locations that are not known at link time. The XAP3 also supports systems where the same code is shared between multiple processes, with each process having its own data.

Position-independent static data is supported using GP-relative addressing with the `mov.g` instruction. Code produced by the compiler is position independent with PC-relative addressing being used for code and constants. Load-time fixups are necessary for certain constructs, typically involving the address-of operator on statically allocated data.

Absolute addressing can be used to access code and data at fixed addresses

# 4 Instruction Set Overview

## 4.1 Summary of Assembler Syntax

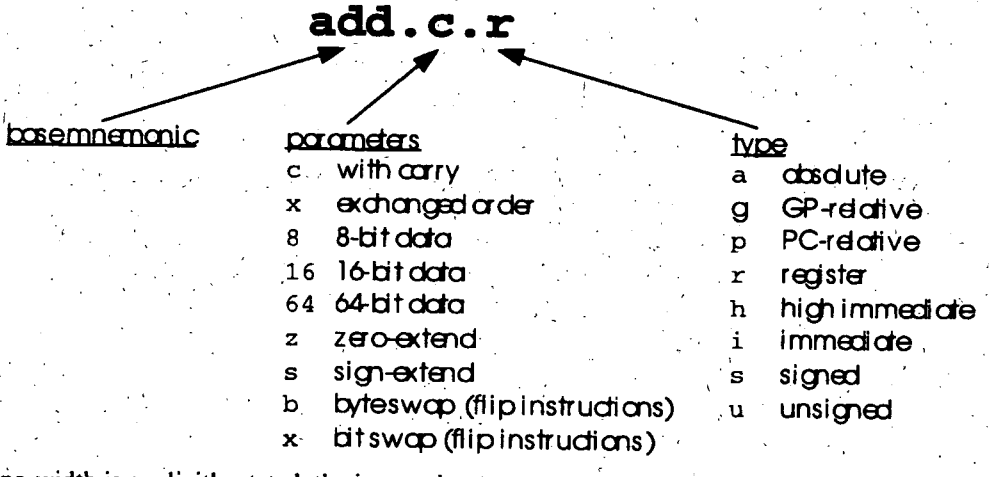
The assembler is case-sensitive. Instruction mnemonics, registers, number prefixes and directives must be in lower case.

Instructions are terminated by an end-of-line. Instructions cannot span more than one line. Spaces and tabs are treated as white space that delimits assembler tokens.

Full details of the assembler syntax are in the XAP3 GNU Binutils (Assembler, Linker and Utilities) Manual.

### 4.1.1 Instruction mnemonics

XAP3 instruction mnemonics have a regular structure, consisting of a base mnemonic, optional parameters and an optional type.




If no width is explicitly stated, the instruction operates on 32-bit data.

### 4.1.2 Operands

Operands are separated by commas.

### 4.1.3 Registers

Register names are lower case and are prefixed by %. XAP3 register names do not pollute the C/C++ namespace.

### 4.1.4 Register Lists

Some instructions take a list of registers as an operand. A RegList can take one of three forms:

- A list of separate registers – `{%r1, %r5, %r8, %r11}`
- A range of registers – `{%r1-%r5}`
- Any mixture of the two forms – `{%r1-%r5, %r12}`, or `{%r1-%r3, %r4, %r8-%r12}`

The registers in the RegList must be from low to high. A register may not appear twice in the RegList.

### 4.1.5 Comments

'C' and C++ style comments are accepted:

```
/*
This is a block comment
*/
and.i %r1, %r2, #0x0000FFFF  // This is a line comment
```

### 4.1.6 Number formats

Immediate values can be entered in decimal, hexadecimal or binary. The assembler converts all immediates to 32 bits and then rejects any immediate value which cannot be represented in an instruction.

#### *Decimal*

Decimal number require no prefix and have an optional sign.

#### *Hexadecimal*

Hexadecimal numbers have an 0x prefix. To enter a negative number in hexadecimal, two approaches can be used:

- Explicitly define all 32 bits – for example, 0xFFFFFFFE.
- Use the unary – operator – for example, -0x2.

The characters a-f and A-F can be used in hexadecimal numbers.

### Binary

Binary numbers have an 0b prefix. To enter a negative number in binary, two approaches can be used:

- Explicitly define all 32 bits – for example, 0b11111111111111111111111111111110.
- Use the unary – operator – for example, -0b10

### Examples of valid numbers

| Assembler syntax | Decimal equivalent |
|---|---|
| 0x1234 | 4660 |
| 0xFFFF4321 | -48351 |
| -0x1234 | -4660 |
| -0xFFFF4321 | 48351 |
| 0b10 | 2 |
| 0b11111111111111111111111111111110 | -2 |
| -0b10 | -2 |
| -0b11111111111111111111111111111110 | 2 |

## 4.1.7 Labels

Labels are case sensitive. Labels start in column 1 and end with a colon. For example:

```
// The start of the program
start:

// This is a different label
Start:
```

## 4.1.8 Directives

Directives start with a period (".") as the first non-blank character of the line. For example:

```
.org 0x1000
.file "Initialisation Code"
.text
```

Full details of the directives are in the XAP3 GNU Binutils (Assembler, Linker and Utilities) Manual.

# 4.2 Immediate Encoding

Many instructions take an immediate value. The width of the immediate bitfield in the instruction varies but is always less than 32 bits. The assembler encodes each immediate operand to generate the correct bitfield for the XAP3 instruction. The rules for performing

the encoding vary according to the instruction type. If the encoding is not possible, the assembler generates an error.

The assembler syntax for immediates in instructions is not affected by these rules. Immediate operands can be 32-bit numbers. However, the encoding rules result in only a subset of the $2^{32}$ possible immediates being valid for each rule.

The five immediate encoding rules are:

| Instruction types | Encoding type | Description of immediate encoding rule |
|---|---|---|
| .i<br>.p<br>.a | no encoding | The immediate operand is entered directly into the bitfield in the instruction. |
| | "s" encoding | A slice of bits from the immediate operand is entered into the bitfield in the instruction.<br>XAP3 interprets the bitfield as a signed number and sign-extends it to 32 bits before use. |
| | "u" encoding | A slice of bits from the immediate operand is entered into the bitfield in the instruction.<br>XAP3 interprets the bitfield as an unsigned number and zero-extends it to 32 bits before use. |
| .h | no encoding | The immediate operand is entered directly into the bitfield in the instruction. |
| | "h" encoding | A slice of bits from the immediate operand starting at bit 31 is entered into the bitfield in the instruction.<br>XAP3 extends the bitfield to 32 bits by copying the right-most bit of the bitfield downwards. |

The encoding type is indicated in the encodings for each instruction.

Immediates of zero should be replaced by `%r0` for maximum efficiency.

Examples of each type of encoding follow.

### 4.2.1 .i instructions – no immediate encoding

The `cmp.8*.i` and `cmp.16*.i` instructions fall into this category.

The `cmp.8*.i` instruction specifies an immediate operand and the ALU performs an 8-bit compare against bits [7:0] of Rs. The immediate operand can be signed or unsigned. Valid operands are therefore in the range 0 to 255, or -128 to +127.

The `cmp.16*.i` instruction specifies an immediate operand and the ALU performs an 16-bit compare against bits [15:0] of Rs. The immediate operand can be signed or unsigned. Valid operands are therefore in the range 0 to 65535, or -32768 to +32767.

### 4.2.2 .i instructions – "s" encoding

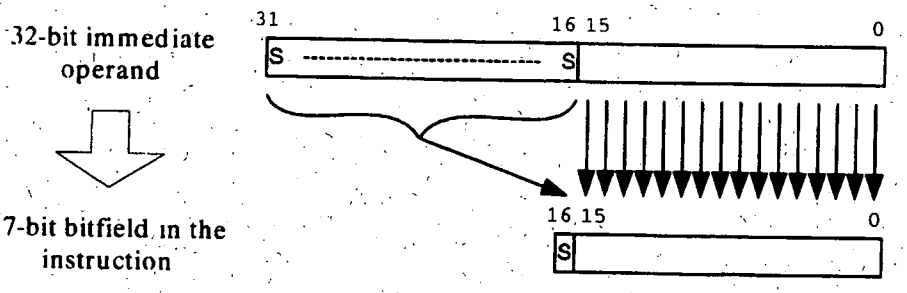
This is indicated by an "s" in the instruction bitfield.

The left-most bit in the instruction bitfield is treated as a sign bit. This significance of this bit defines the valid range of the immediate operand. As an example, if the significance of the left-most bit is $2^8$, the range of valid immediates operands is -256 to +255.

Some instructions do not encode the low order bits of the immediate operand. These must be zero in the operand.

### *Example of "s" encoding in a 17-bit immediate .i instruction*

This is used in the 32-bit encoding for the `add.i` instruction and similar instructions.

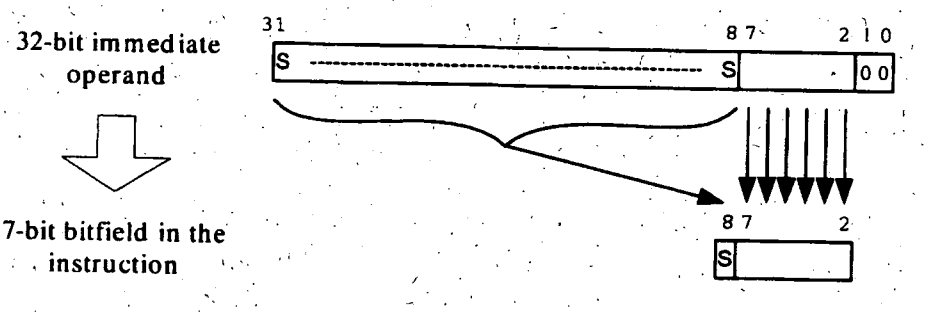



The valid range for immediate operands is -65536 to +65535. Valid instructions are:

```
add.i %r1, %r2, #65535
add.i %r1, %r2, #-65536
add.i %r1, %r2, #0xFFFF0000 // -65536
add.i %r1, %r2, #0x0000FFFF // +65535
add.i %r1, %r2, #0xFFFF     // +65535
```

### *Example of "s" encoding in a 7-bit immediate .i instruction*

This is used on the 16-bit encoding for the `add.i Rd, %r15` instruction. The instruction encodes bits [8:2]s of the immediate. Bits 1 and 0 are not specified in the instruction amd must be zero in the immediate operand.



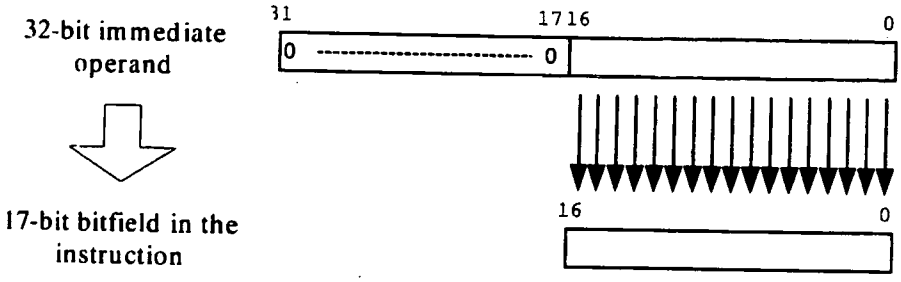

The valid range for immediate operands is -256 to +252. Valid instructions are:

```
add.i %r1, %r15, #-256
add.i %r1, %r15, #252
add.i %r1, %r15, #0xFFFFFF00 // -256
add.i %r1, %r15, #0x000000FC // +252
add.i %r1, %r15, #0xFC       // +252
```

### 4.2.3 .i instructions – "u" encoding

This is indicated by a "u" in the instruction bitfield. "u" encoding also applies to offsets in the push and pop instructions.

The immediate operand is treated as unsigned. The significance of the left-most bit in the instruction bitfield defines the valid range of the immediate operand. As an example, if the significance of the left-most bit is $2^{16}$, the range of valid immediates operands is 0 to 131071.




As with the "s" encoding rule, some instructions do not encode the lowest bits of the operand. These must be zero.

Examples of valid instructions are:

```
ld.16z.i %r1, @(2, %r15)      // This gives a 16-bit encoding
stm.x    {%r1-%r6}, @(4, %r15)
```

### 4.2.4 .h instructions – no immediate encoding

The `cmp.16*.h` instuctions fall into this category

The `cmp.16*.h` instruction specifies an immediate operand and the ALU performs an 16-bit compare against bits [31:16] of Rs. Valid operands are in the range 0xFFFF0000 to 0xFFFFFFFF. The least-significant 16 bits in the immediate operand are not encoded in the instruction and must be zero.

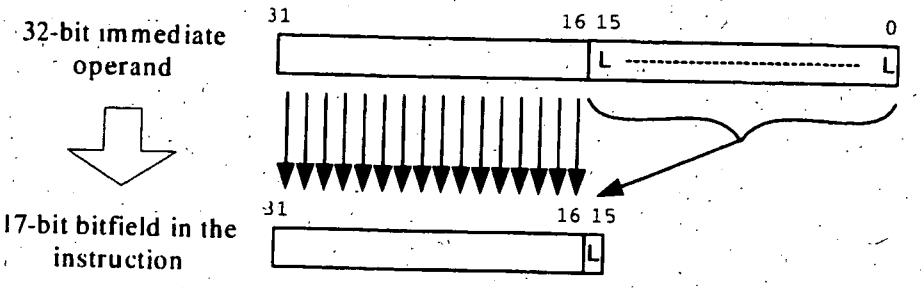
Examples of valid instructions are:

```
cmp.16.h  %r1,  #0xFFFF0000
cmp.16c.h %r2, #0xAAAA0000
```

### 4.2.5 .h instructions – "h" encoding

This is indicated by an "h" in the instruction bitfield.

The instruction bitfield encodes bits [31:15] of the immediate operand. Bits [14:0] of the immediate operand must be the same as bit 15.




The valid range for immediate operands is 0xFFFF0000 to 0xFFFFFFFF. Valid instructions are:

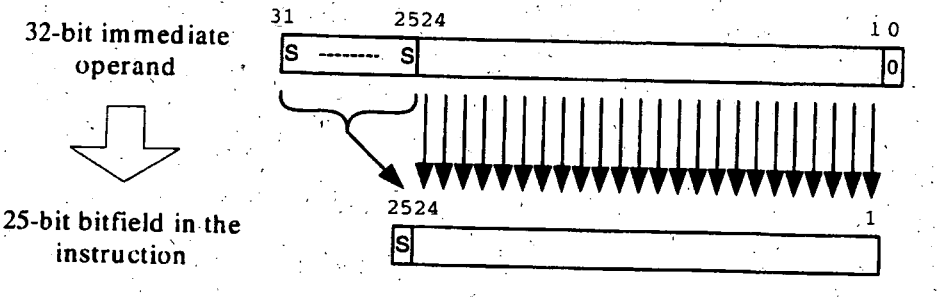
```
add.h %r1, %r2, #0xFFFF0000 // +4294901760 or -65536
add.h %r1, %r2, #0x00FF0000 // +16711680
add.h %r1, %r2, #0x00FFFFFF // +16777215
```

### 4.2.6 .p instructions – "s" encoding

`bra.p`, `bsr.p` and `mov.p` take a 25-bit PC-relative offset. The offset is signed and is sign-extended to 32 bits before use. As all instructions start on a 16-bit boundary, bit 0 of the offset must be zero and is not encoded in the instruction.

#### *Example of "s" encoding in a 25-bit immediate .p instruction*

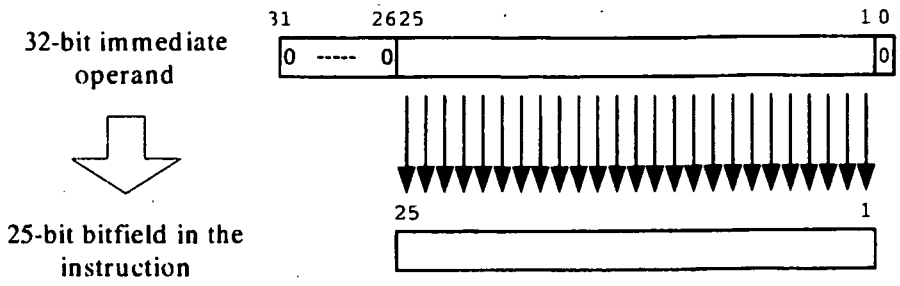
This is used in the 32-bit encodings for the `bra.p`, `bsr.p` and `mov.p` instructions.




The valid range for operands is -32MByte to +32MByte. The operand represents an offset relative to the current PC.

### 4.2.7 .a instructions – "u" encoding

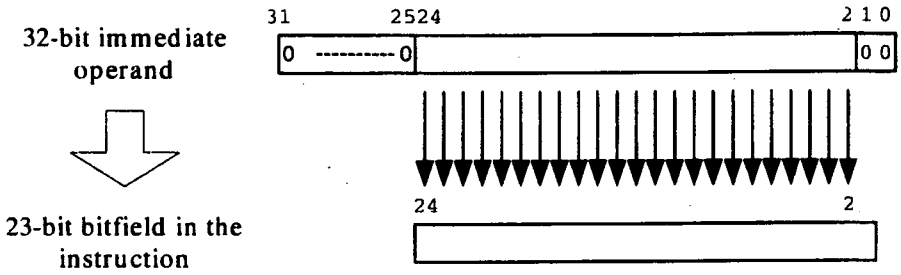
`bra.a` and `bsr.a` take a 25-bit unsigned absolute address. As all instructions start on a 16-bit boundary, bit 0 of the offset must be zero and is not encoded in the instruction.




The valid range for operands is 0 to 64MByte.

### 4.2.8 mov.g – "u" encoding

`mov.g` takes a 23-bit signed unsigned offset. The offset is the start address of a data object relative to GP. As all data objects start on a 32-bit boundary, bits [1:0] of the offset must be zero and are not encoded in the instruction.




The valid range for operands is 0 to 32MByte.

### 4.2.9 Register or small immediate values

Some of the instructions specify either a small immediate or a register (e.g. movm) as an operand. The instruction encoding of each operand uses five bits: one bit selects between register and immediate, four bits specify the register or the immediate.

To specify an immediate value of zero, use register R0. An immediate operand of -1 is encoded in the instruction bitfield as zero. Thus a four-bit immediate bitfield can represent numbers -1, 1, 2, ..., 15.

### 4.2.10 Examples of invalid immediates in instructions

| Invalid Instruction | Reason |
|---|---|
| `add.i %r1, %r2, #0x00012345` | Not all bits 31:16 are the same. |
| `add.h %r1, %r2, #0x12345000` | Not all bits 15:0 are the same. |

## 4.3 Instruction Encoding

XAP3 instructions are either 32 bits or 16 bits long. The entire instruction set can be represented by the 32-bit instructions. The most commonly used instructions are available as 16-bit encodings. The 16-bit encodings sometimes support a smaller range of immediates or registers 16-bit and 32-bit instructions can be freely mixed.

The assembler syntax for both 32-bit and 16-bit instructions is identical The assembler selects the smallest encoding.

An exception is `bra.p`. This instruction is commonly used after the `bcase.2` or `bcase.4` instructions, which are commonly used to implement C/C++ switch statements. In this situation it is necessary to force the assembler to generate 16-bit or 32-bit `bra.p` instructions. There are therefore specific variants of the `bra.p` instruction:

- `bra.p.2` forces the assembler to generate a 16-bit encoding
- `bra.p.4` forces the assembler to generate a 32-bit encoding

## 4.4 Addressing Modes

The XAP3 has a "load/store" architecture. There are a small number of instructions which transfer memory data to and from registers. Most instructions operate on registers directly.

The only XAP3 instructions which can access memory are:

- `ld` and its variants
- `st` and its variants
- `push` and its variants
- `pop` and its variants
- `swap`

There are two addressing modes for memory operations in `ld` and `st` instructions – displacement addressing and indexed addressing.

### *Displacement addressing*

Displacement addressing is indicated by a .i suffix to the instruction. The instruction specifies a base address in a register and a literal offset. The memory address is calculated as the sum of the two. For example:

```
mov.i %r2, #0x1000
ld.i  %r1, @(0x20, %r2)
```

This will load R1 from memory address 0x1020.

### *Indexed addressing*

Displacement addressing is indicated by a .r suffix to the instruction. The instruction specifies two registers, giving a base address and an offset. The memory address is calculated as follows:

| Instruction | Calculated memory address |
|---|---|
| `ld.8s.r Rd, @(Rx, Ra)`<br>`ld.8z.r Rd, @(Rx, Ra)` | Rx + Ra |
| `ld.16s.r Rd, @(Rx, Ra)`<br>`ld.16z.r Rd, @(Rx, Ra)` | 2*Rx + Ra |
| `ld.r Rd, @(Rx, Ra)` | 4*Rx + Ra |

# 5 Instruction Groups

The following sections provide an overview of each group of instructions.

## 5.1.1 Branches

### *Unconditional branches*

The XAP3 has three unconditional branches. The branch address can be an absolute address (`bra.a`), PC-relative (`bra.p`) or contained in a register (`bra.r`).

The same three operations are available when branching to a subroutine (`bsr.a`, `bsr.p` and `bsr.r`).

### *Conditional branches*

All conditional branches are relative to the program counter. A full set of conditions is supported.

### *Case branches*

The `bcase.2` and `bcase.4` instructions implement a computed-goto which can be used to implement the C/C++ switch statement.

## 5.1.2 Load and Store

### *Single memory transfers*

Loads and stores are available in 8, 16 and 32-bit versions and with displacement or indexed addressing. For 8 and 16-bit data, the load instructions also specify whether the data is to be sign-extended to zero-extended to 32 bits.

If a load of register R0 is specified, the specified memory address is read and the flags updated, but the memory data is discarded.

### Multiple memory transfers

The ldm and stm instructions allow a selection of registers to be transferred to or from memory in a single instruction. The instructions are atomic and are therefore useful for context switches and procedure calls. All transfers are 32 bits wide.

Three variants are available:

| Instructions | Brief description |
|---|---|
| `ldm, stm` | Operate on the current set of registers. These instructions are intended for code running in User Mode. |
| `ldm.x, stm.x` | As `ldm` and `stm`, but the order of the registers in memory is reversed. |
| `ldm.u, stm.u` | Operate on the User Mode registers. These instructions cannot be used in User Mode and are useful for task switches by an operating system. |

The selected registers are stored sequentially in memory. The following table demonstrates the order, assuming all registers are selected:

| Memory address | ldm, stm<br>ldm.u, stm.u | ldm.x, stm.x |
|---|---|---|
| Highest address | R15 | R0 |
| | R14 | R1 |
| | ... | ... |
| | R1 | R14 |
| Lowest address | R0 | R15 |

The FLAGS register is updated on the basis of the highest numbered register in the load.

`ldm`, `stm`, `ldm.x` and `stm.x` can be used in all four operating modes. `ldm.u` and `stm.u` can only be used in the privileged modes. The registers used by the instructions change according to the processors operating mode and are:

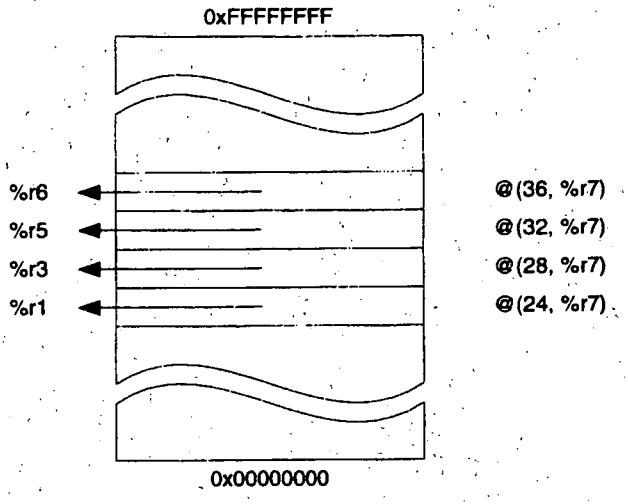
| | User Mode | Supervisor Mode | Interrupt Mode | NMI Mode |
|---|---|---|---|---|
| `ldm, stm` | User Mode registers | Supervisor Mode registers | Interrupt Mode registers | NMI Mode registers |
| `ldm.x, stm.x` | User Mode registers | Supervisor Mode registers | Interrupt Mode registers | NMI Mode registers |
| `ldm.u, ldm.u` | Throws an `IllegalInstruction` exception | User Mode registers | User Mode registers | User Mode registers |

### ldm and stm – example code

The instruction:

```
ldm (%r1, %r3, %r5, %r6), @(24, %r7)
```

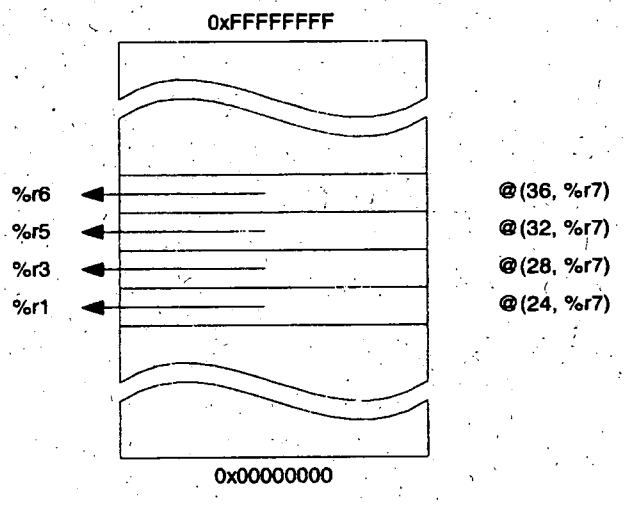
would be executed as:




The instruction:

```
stm (%r1, %r3, %r5, %r6), @(24, %r7)
```

would be executed as:




### *Swap*

The swap instruction performs an atomic swap between a register and a memory location. This is a useful instruction for implementing sempahores in operating systems. Refer to section "Semaphore" for an example.

## 5.1.3 Push and Pop

The push and pop instructions enable very compact code for stack operations and function entry and exit. Multiple registers can be pushed or popped in a single instruction. The instructions are atomic.

The stack pointer is R15. The stack grows downwards in memory. Lower numbered registers are stored at lower stack addresses.

### push

The push instruction pushes a selection of registers on to the stack. The flags are not updated. An additional operand adjusts the stack pointer downwards to create a stack frame for the callee function's automatic variables.

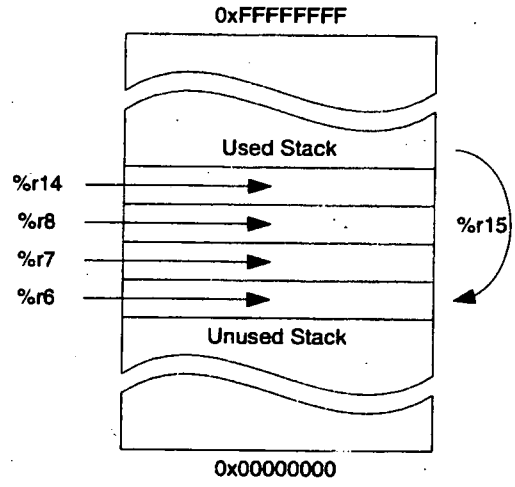
### push – example 1

```
push {%r6, %r7, %r8, %r14}, #0
```

This is equivalent to the following sequence of instructions.

```
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r14, @(0, %r15)   // Store R14 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r8, @(0, %r15)    // Store R8 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r7, @(0, %r15)    // Store R7 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r6, @(0, %r15)    // Store R6 to stack memory
```

This is illustrated in the diagram below.



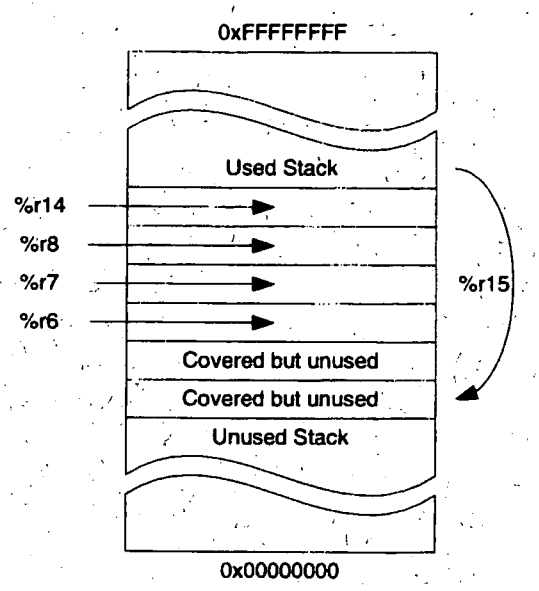

### push – example 2

```
push    {%r6, %r7, %r8, %r14}, #8
```

This is equivalent to the following sequence of instructions.

```
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r14, @(0, %r15)   // Store R14 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r8, @(0, %r15)    // Store R8 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r7, @(0, %r15)    // Store R7 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r6, @(0, %r15)    // Store R6 to stack memory
add.i   %r15, %r15, #-8    // Decrease SP by 8 giving
                           //    2 words of stack free
                           //    for other purposes
```

This is illustrated in the diagram below.




### *push.r*

The `push.r` instruction implements part of the calling convention for C functions. Its operation is:

- Push selected R6-R14 registers to the stack.
- Copy selected R1-R5 registers to R6-R14. There must be more registers specified in the R6-R14 range than in the R1-R5 range.

The flags are updated according to the last copy operation. If there is no copy operation (none of R1-R5 are selected), the flags remain unchanged.

A with `push`, an additional operand adjusts the stack pointer downwards to create a stack frame for the callee function's automatic variables.

### *push.r – example*

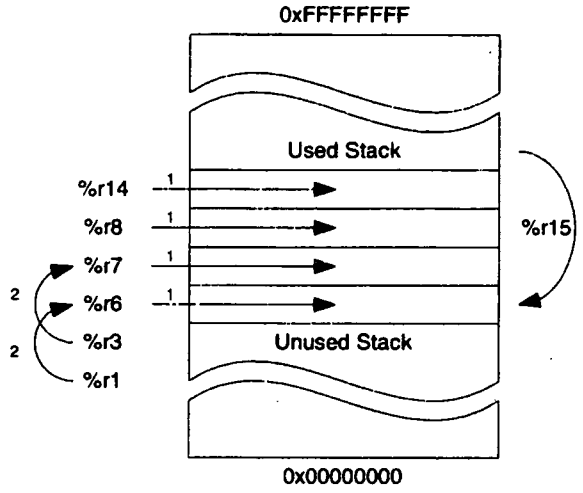
```
push.r {%r1, %r3, %r6, %r7, %r8, %r14}, #0
```

This is equivalent to the following sequence of instructions.

```
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r14, @(0, %r15)   // Store R14 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r8, @(0, %r15)    // Store R8 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r7, @(0, %r15)    // Store R7 to stack memory
add.i   %r15, %r15, #-4    // Decrease SP for 1 register
st      %r6, @(0, %r15)    // Store R6 to stack memory
mov.r   %r7, %r3           // Copy R3 to R7 (2nd of R6-R14
                           //      in register list)
mov.r   %r6, %r1           // Copy R1 to R6 (1st of R6-R14
                           //      in register list)
                           //      and set the Z, N flags
```

```
                                //      accordingly.
```

This is illustrated in the diagram below.




## *pop*

The pop instruction pops selected registers off the stack. Registers R6-R14 can be specified. It is not normally necessary to pop R1-R5 from the stack as they are not preserved over a function call, but this can be done with `ldm` if needed.

pop takes two other operands:

- The return value for the function. This is loaded into R1. The return value can be an immediate or another register. The flags are updated according to a comparison of the return value with zero.
- An operand to adjust the stack pointer upwards to remove the stack space allocated for the callee function's automatic variables.

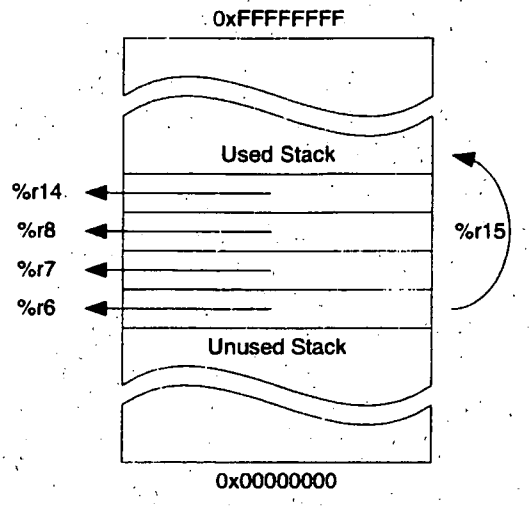
## *pop – example 1*

```
pop     #7, {%r6, %r7, %r8, %r14}, #0
```

This is equivalent to the following sequence of instructions :

```
mov.i   %r1, #7             // Copy 7 to R1 (return register)
ld      %r6, @(0, %r15)     // Load R6 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r7, @(0, %r15)     // Load R7 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r8, @(0, %r15)     // Load R8 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r14, @(0, %r15)    // Load R14 from memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
cmp.i   %r1, #0             // Set flags
```

This is illustrated in the diagram below.



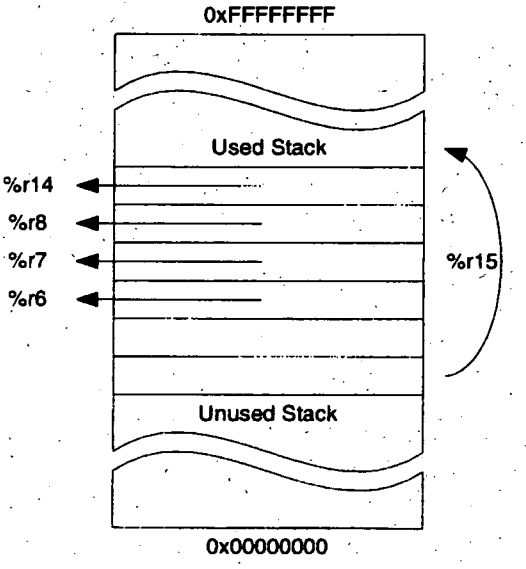

### *pop – example 2*

```
pop     %r9, {%r6, %r7, %r8, %r14}, #8
```

This is equivalent to the following sequence of instructions :

```
mov.r   %r1, %r9          // Copy R9 to R1 (return register)
add.i   %r15, %r15, #8    // Increase SP by 8
ld      %r6, @(0, %r15)   // Load R6 from stack memory
add.i   %r15, %r15, #4    // Increase SP for 1 register
ld      %r7, @(0, %r15)   // Load R7 from stack memory
add.i   %r15, %r15, #4    // Increase SP for 1 register
ld      %r8, @(0, %r15)   // Load R8 from stack memory
add.i   %r15, %r15, #4    // Increase SP for 1 register
ld      %r14, @(0, %r15)  // Load R14 from memory
add.i   %r15, %r15, #4    // Increase SP for 1 register
cmp.i   %r1, #0           // Set flags
```

This is illustrated in the diagram below.



### *pop.ret*

`pop.ret` performs the same operations as pop but additionally returns from the function by copying the link register (R14) into the program counter.

### *pop.ret – example 1*

```
pop.ret #-1, {%r6, %r7, %r8, %r14}, #0
```

This is equivalent to the following sequence of instructions :

```
mov.i   %r1, #-1            // Copy -1 to R1 (return register)
ld      %r6, @(0, %r15)     // Load R6 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r7, @(0, %r15)     // Load R7 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r8, @(0, %r15)     // Load R8 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r14, @(0, %r15)    // Load R14 from memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
cmp.i   %r1, #0             // Set flags
bra.r   %r14                // Return to address in R14 (LR)
```

### *pop.ret example 2*

```
pop.ret %r11, {%r6, %r7, %r8, %r14}, #8
```

This is equivalent to the following sequence of instructions :

```
mov.r   %r1, %r11           // Copy R9 to R1 (return register)
add.i   %r15, %r15, #8      // Increase SP by 8
ld      %r6, @(0, %r15)     // Load R6 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r7, @(0, %r15)     // Load R7 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r8, @(0, %r15)     // Load R8 from stack memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
ld      %r14, @(0, %r15)    // Load R14 from memory
add.i   %r15, %r15, #4      // Increase SP for 1 register
cmp.i   %r1, #0             // Set flags
bra.r   %r14                // Return to address in R14 (LR)
```

## 5.1.4 Move

The mov instructions provide various methods for loading one or more registers with literal values or other runtime constants.

### *mov.p – PC-relative*

The instruction specifies a signed offset relative to the program counter, in the range -32MByte to +32MByte. The least-significant bit must be zero and only the most-significant 25 bits are encoded in the instruction.

### *mov.g – GP-relative*

The instruction specifies an unsigned offset relative to the GP register, in the range 0 to +32MByte. The least-significant two bits must be zero and only the most-significant 23 bits are encoded in the instruction

### *mov.i and mov.h – immediates*

Low or high immediates can be loaded into a register. To load a 32-bit immediate, two instructions are needed:

```
mov.i %r1, #0x5678          // R1 becomes 0x00005678
or.h  %r1, %r0, #0x12340000 // R1 becomes 0x12345678
```

### *mov.r – register to register*

`mov.r` allows a single register-to-register copy.

### *movm and movm.x – move multiple*

The movm instructions copy registers or immediates to registers R1-R5. Any mixture fof register or immediates as source values can be used. Immediates in the range -1, 1, 2, .. 15 can be used. To set a register to zero, specify R0 as the source register.

movm loads R1 first, R5 last. `movm.x` loads R5 first, R1 last. The difference is significant for register-to-register copies, where the same registers are used as source and destination.

These instructions are useful for preparing registers for function calls.

## 5.1.5 ALU operations

The ALU supports a diverse set of logical and arithmetic operations. The general operands are a destination register, a source register, and a third operand which may be either a third operand or an immediate value.

### *Divide and remainder instructions*

Division is performed FORTRAN-style:

- Both operands are made unsigned
- The division is performed
- The sign of the result is corrected if necessary

The div and rem instructions work together such that

`((a/b)*b) + (a%b) = a`

This is true for `div.s` with `rem.s` and `div.u` with `rem.u`.

`div.s` trucates towards zero:

```
5 / 2 =  2    5 % 2 =  1
```

```
-5 /-2 = -2   -5 %-2 = -1
-5 /-2 =  2   -5 %-2 = -1
```

### *Divide by zero*

Both signed and unsigned divide and remainder instructions throw a `DivideByZero` exception if the denominator is zero and the processor is in User Mode.

When a divide by zero is attempted in one of the privileged modes, the division does not take place and the destination register remains unchanged.

The flags are updated according to whether the operation was signed or unsigned. An unsigned operation sets the C flag. A signed operation sets the V flag.

## 5.1.6 Compare operations

The compare instructions are used to set the processors's flags. These are used by the conditional branch instructions.

The 8-bit variants of the compare instructions operate on the bottom 8 bits of the operands only. The 16-bit .i variants operate on the bottom 16 bits of the operands only. The 16-bit .h variants operate on the top 16 bits of the operands only.

A 32-bit compare against a 32-bit immediate can be performed with two cmp instrictions. Fexample, to compare R4 with 0x43211234:

```
cmp.16.i      %r1, #0x00001234
cmp.16c.h     %r1, #0x43210000
```

## 5.1.7 Shift and rotate

The XAP3 has signed (arithmetic) and unsigned (logical) left and right shift instructions. A register can be shifted up to 31 places left or right in a single instruction. For left shifts, the bit shifted out of bit 31 is copied to the carry flag. For right shifts, the bit shifted out of bit 0 is copied to the carry flag.

The `rotatel` instruction performs a rotate left of up to 31 places. The rotate does not circulate through the carry bit.

The rlinm (rotate left immediate and mask) instruction is a powerful instruction for shifting words and performing bit operations such as bitfield extraction. There are both zero-extending and sign-extending versions of this instruction.

Shifts and rotates by an immediate value are implemented using the rlinm instructions.

## 5.1.8 DSP instructions

The XAP3 provides a number of powerful instructions for complex data manipulation:

| Instruction | Description |
|---|---|
| `flip.*` | Six instructions useful for endianness conversion and selecting bit slices |

| | |
|---|---|
| | from a 64-bit input structure. |
| `abs.r` | Calculates the absolute value of a signed number |
| `msbit.r` | Finds the most significant one in the word. This can be used to simplify the process of normalisation in floating point arithmetic |
| `acc64.r` | Accumlate into a 64-bit accumulator. |
| `shiftl.64`, `shiftr.64` | Left or right shift by up to 63 places in a 64-bit accumulator |

### 5.1.9 Miscellaneous instructions

#### *Interrupt-related*

`irqd` disables interrupts by clearing the E bit in the FLAGS register. `irqe` enables interrupts by setting the E bit. `irqs` returns the current interrupt status.

`rtie` is used to return from an interrupt or exception.

None of these instructions are permitted in User Mode.

#### *Traps*

`trap.r` and `trap.i` throw a Trap exception. This allows User Mode code to call Supervisor Mode code. The trap instructions are executed as a nop in the privileged modes.

#### *Sleeping and no-operation*

`sleepnop` and `sleepsif` put the XAP3 into the NOP Sleep and SIF Sleep states, respectively. The hardware WAKE_UP signal is necessary to restart the processor. In the SIF Sleep state (`sleepsif`), SIF cycles are permitted. In the NOP Sleep state (`sleepnop`), SIF cycles are not permitted. Neither instruction is permitted in User Mode.

`nop` is the no operation instruction.

#### *Debug-related*

`brk` implements a software breakpoint. If the processor is in User Mode and the B bit in the FLAGS register is set, the Break exception is thrown. Otherwise, if the processor is in debug mode, the processor is stopped.

`halt` stops the processor. This instruction is not permitted in User Mode.

`sif` allows the processor to perform a SIF cycle. This instruction is not permitted in User Mode.

`print.r` causes xIDE to print a character in the debug window. This is useful for putchar emulation during simulation.

The `ver` and `lic` instructions return information about the XAP3 core.

### *Operations on special registers*

movr2s and movs2r provide access to the special registers.

movr2b and movb2r provide access to the breakpoint registers.

## 5.2 Common Code Sequences

### 5.2.1 Function prologue and epilogue

For a function with no stack requirement, the simplest function epilogue is:

```
cmp.i  %r1, #0  // Sets the flags (as per calling convention)
bra.r  %r14     // Return to caller
```

The cmp instruction is not needed if the function returns a void.

For arbitrary functions, the prologue and epilogue code varies according to:

- Whether the function is a leaf function or not. A leaf function does not make any function calls itself.
- The amount of stack needed for the function's automatic variables, if any.

### 5.2.2 Nested interrupt prologue and epilogue

For nested interrupts, additional processor state must be saved and restored. A suitable prologue is:

```
push   {%r1-%r14}, #4          // Save registers and allocate
                               //  one word on the stack
movs2r %r1, %flags_i           // Get SFLAGS_I
st     %r1, @(0,%r15)          //    and save on the stack
irqe                           // Re-enable interrupts
```

Then on exit we restore the original registers and flags:

```
irqd                           // Disable interrupts
ld     %r1, @(0,%r15)          // Get SFLAGS_I from stack
movr2s %flags_i, %r1           // Restore SFLAGS_I
ldm    {%r1-%r5}, @(4, %r15)   // Restore R1-R5
pop    {%r6-%r14}, #24         // Restore registers
                               //    and free stack space
rtie                           // Return from interrupt
```

The exact register lists for the push and pop can be tailored for specific handlers to minimise stack usage and interrupt response time.

NMI handlers that are expected to return control to the interrupted process may use similar entry and exit code, with the omission of the irqe and irqd instructions

### 5.2.3 Semaphore

The swap instruction can be used to implement a semaphore without needing to disable interrupts. Two assembly functions are needed – one to lock (acquire) a semaphore, and a second to release it.

`osXSimpleGetSem()` attempts to acquire a semaphore. The `swap.r` instruction swaps-in the "locked" value, and gives the current semaphore state in R3.

```
    .text

    .globl osXSimpleGetSem, osXSimpleReleaseSem

// osXSimpleGetSem
// ===============
// Description
//    Try to acquire the semaphore specified in R1.  Returns immediately
//    if the semaphore is already locked.  There is no range checking on
//    the semaphore number in R1.
// Entry:
//    R1 = semaphore number (0..n)
//    R2 = base address of semaphore array
// Exit:
//    R1 = 0 indicates the semaphore was acquired successfully
//    R1 = 1 indicates another task already has the semaphore
//
osXSimpleGetSem:
    mov.i  %r3, #1
    swap.r %r3, @(%r1, %r2)
    mov.r  %r1, %r3
    cmp.r  %r1, #0
    bra.r  %r14
```

`osXSimpleReleaseSem()` uses the `swap.r` instruction to set the semaphore state to "unlocked". The value retrieved from the semaphore should be 1 but this simple code does not check that the semaphore was locked on entry; nor that it was locked by the current process.

```
    .text

    .globl osXSimpleReleaseSem

// osXSimpleReleaseSem
// ===================
// Description
//    Releases a semaphore.  No checking is performed (eg, that this
//    task acquired the semaphore successfully in the first place).
// Entry:
//    R1 = semaphore number (0..n)
//    R2 = base address of semaphore array
// Exit:
//    R1 = undefined
//
osXSimpleReleaseSem:
    mov.i  %r3, #0
    swap.r %r3, @(%r1, %r2)
    bra.r  %r14
```

These functions can be used from 'C' as follows:

```
extern int  osXSimpleGetSem    (int, int *);
extern void osXSimpleReleaseSem(int, int *);

int int osXSimpleSemPool[NUM_SEMAPHORES];

{
      if ( 0 == osXSimpleGetSem(MY_SEMAPHORE, osXSimpleSemPool) )
      {
          // We have the semaphore
          ...
          osXSimpleReleaseSem(MY_SEMAPHORE);
      }
}
```

### 5.2.4 Initialisation of User Mode registers

Code running in Supervisor Mode can initialise the User Mode registers before starting a User Mode task. R1, R2, R14 (LR), R15 (SP) and GP_U need explicit initialisation. Registers R3-R13 and the FLAGS register are shared with Supervisor Mode and can be initialised with normal instructions.

```
inittab: .word 0x11111111   // R1
         .word 0x22222222   // R2
         .word killfunction // to kill the task if it ends
         .word stacktop     // initial value for R15

osXInitUserRegs:
   // Initialise R1, R2, R14 (LR) and R15 (SP)
   mov.p  %r1, #inittab
   ldm.u  {%r1, %r2, %r14, %r15}, @(0, %r1)

   // Initialise GP_U
   mov.p  %r1, #usergp  // Probably zero in most applications
   movr2s %gp_u, %r1

   // Return to caller
   bra.r  %r14
```

### 5.2.5 DSP Example

This example demonstrates the use of the 64-bit accumulator and the `abs.r` instruction.

```
// int ComputeAbsSum( int * p, int nelems )
// ========================================
// Description
//   Computes the sum of the absolute values of all entries in
//   the array pointed to by p of nelems integer elements.
//
//   Note that there must be atleast one element in the array.
//

ComputeAbsSum:
    .globl ComputeAbsSum
```

```
    push        {%r8,%r9}, #0
    mov.i       %r8, #0             // clear 64-bit accumulator {R9, R8}
    mov.i       %r9, #0

L:  ld.r        %r3, @(%r0, %r1)    // get value from array
    abs.r       %r3, %r3            // compute absolute value
    acc.64.r    %r8, %r3            // add to the 64-bit accumulator
    add.i       %r1, %r1, #4        // bump the data pointer
    add.i       %r2, %r2, #-1       // and size counter
    bne         L                   // keep going if more data

    pop.ret     %r8, {%r8, %r9}, #0
```

## 5.3 Instructions Grouped by Function

### 5.3.1 Branches

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Unconditional Branch | bra.a | Branch, absolute | ---- |
| | bra.p<br>bra.p.2<br>bra.p.4 | Branch, PC-relative | ---- |
| | bra.r | Branch, register | ---- |
| Unconditional Branch Subroutine | bsr.a | Branch to subroutine , absolute | ---- |
| | bsr.p | Branch to subroutine, PC-relative | ---- |
| | bsr.r | Branch to subroutine, register | ---- |
| Conditional Branch | beq | Branch if equal | ---- |
| | bne | Branch if not equal | ---- |
| | bmi | Branch if minus | ---- |
| | bpl | Branch if plus | ---- |
| | bcs | Branch if carry set | ---- |
| | bcc | Branch if carry clear | ---- |
| | bvs | Branch if overflow set | ---- |
| | bvc | Branch if overflow clear | ---- |
| | blt.s | Branch if less than, signed | ---- |
| | bge.s | Branch if greater than, signed | ---- |
| | ble.s | Branch if less than or equal, signed | ---- |
| | bgt.s | Branch if greater than, signed | ---- |
| | blt.u | Branch if less than, unsigned | ---- |
| | bge.u | Branch if greater than or equal, unsigned | ---- |
| | ble.u | Branch if less than or equal, unsigned | ---- |

| | | | |
|---|---|---|---|
| | bgt.u | Branch if greater than, unsigned | ---- |
| Case Branch | bcase.2 | Case branch into 4-byte table | ---- |
| | bcase.4 | Case branch into 2-byte table | ---- |

### 5.3.2 Load and Store

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Load (displacement) | ld.i | Load, displacement | ZN-- |
| | ld.16s.i | Load, 16-bit, sign-extend, displacement | ZN-- |
| | ld.16z.i | Load, 16-bit, zero-extend, displacement | ZN-- |
| | ld.8s.i | Load, 8-bit, sign-extend, displacement | ZN-- |
| | ld.8z.i | Load, 8-bit, zero-extend, displacement | ZN-- |
| Load (indexed) | ld.r | Load, indexed | ZN-- |
| | ld.16s.r | Load, 16-bit, sign-extend, indexed | ZN-- |
| | ld.16z.r | Load, 16-bit, zero-extend, indexed | ZN-- |
| | ld.8s.r | Load, 8-bit, sign-extend, indexed | ZN-- |
| | ld.8z.r | Load, 8-bit, zero-extend, indexed | ZN-- |
| Load multiple | ldm | Load multiple | ZN-- |
| | ldm.x | Load multiple, exchanged order | ZN-- |
| | ldm.u | Load multiple, user | ZN-- |
| Store (displacement) | st.i | Store, displacement | ---- |
| | st.16.i | Store, 16-bit, displacement | ---- |
| | st.8.i | Store, 8-bit, displacement | ---- |
| Store (indexed) | st.r | Store, indexed | ---- |
| | st.16.r | Store, 16-bit, indexed | ---- |
| | st.8.r | Store, 8-bit, indexed | ---- |
| Store multiple | stm | Store multiple | ---- |
| | stm.x | Store multiple, exchanged order | ---- |
| | stm.u | Store multiple, user | ---- |
| Swap | swap.r | Swap register with memory | ZN-- |

### 5.3.3 Push and Pop

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Push | push | Push to stack | ---- |
| | push.r | Push to stack and registers | ZN-- |
| Pop | pop | Pop from stack | ZNCV |
| | pop.ret | Pop from stack and return | ZNCV |

### 5.3.4 Move

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Move | mov.p | Move, PC-relative | ---- |
| | mov.g | Move, GP-relative | ---- |
| | mov.i | Move, immediate | ---- |
| | mov.h | Move, high immediate | ---- |
| | mov.r | Move, register | ---- |
| | movm | Move multiple | ZN-- |
| | movm.x | Move multiple, exchanged order | ZN-- |

### 5.3.5 Conditional Set

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Conditional set | scc.i | Set if carry clear, immediate | ---- |
| | scc.r | Set if carry clear, register | ---- |
| | scs.i | Set if carry set, immediate | ---- |
| | scs.r | Set if carry clear, register | ---- |
| | seq.i | Set if equal, immediate | ---- |
| | seq.r | Set if equal, register | ---- |
| | sge.s.i | Set if greater than or equal, signed, immediate | ---- |
| | sge.s.r | Set if greater than or equal, signed, register | ---- |
| | sge.u.i | Set if greater than or equal , unsigned, immediate | ---- |
| | sge.u.r | Set if greater than or equal , unsigned, register | ---- |
| | sgt.s.i | Set if greater than, signed, immediate | ---- |
| | sgt.s.r | Set if greater than, signed, register | ---- |
| | sgt.u.i | Set if greater than, unsigned, immediate | ---- |
| | sgt.u.r | Set if greater than, unsigned, register | ---- |
| | sle.s.i | Set if less than, signed, immediate | ---- |
| | sle.s.r | Set if less than, signed, register | ---- |
| | sle.u.i | Set if less than or equal, unsigned, immediate | ---- |
| | sle.u.r | Set if less than or equal, unsigned, register | ---- |
| | slt.s.i | Set if less than, signed, immediate | ---- |
| | slt.s.r | Set if less than, signed, register | ---- |
| | slt.u.i | Set if less than, unsigned, immediate | ---- |
| | slt.u.r | Set if less than, unsigned, register | ---- |
| | smi.i | Set if minus, immediate | ---- |
| | smi.r | Set if minus, register | ---- |

| | | | |
|---|---|---|---|
| | sne.i | Set if not equal, immediate | ---- |
| | sne.r | Set if not equal, register | ---- |
| | spl.i | Set if plus, immediate | ---- |
| | spl.r | Set if plus, register | ---- |
| | svc.i | Set if overflow clear, immediate | ---- |
| | svc.r | Set if overflow clear, register | ---- |
| | svs.i | Set if overflow set, immediate | ---- |
| | svs.r | Set if overflow set, register | ---- |

### 5.3.6 ALU operations

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Add | add.c.r | Add, with carry, register | ZNCV |
| | add.h | Add, high immediate | ZNCV |
| | add.i | Add, immediate | ZNCV |
| | add.r | Add, register | ZNCV |
| Subtract | sub.c.r | Subtract, with carry, register | ZNCV |
| | sub.r | Subtract, register | ZNCV |
| | sub.x.h | Subtract, exchanged order, high immediate | ZNCV |
| | sub.x.i | Subtract, exchanged order, immediate | ZNCV |
| Logical | and.h | And, high immediate | ZN-- |
| | and.i | And, immediate | ZN-- |
| | and.r | And, register | ZN-- |
| | or.h | Or, high immediate | ZN-- |
| | or.i | Or, immediate | ZN-- |
| | or.r | Or, register | ZN-- |
| | xor.h | Exclusive-or, high immediate | ZN-- |
| | xor.i | Exclusive-or, immediate | ZN-- |
| | xor.r | Exclusive-or, register | ZN-- |
| Multiply | mult.16s.i | Multiply, 16-bit signed, immediate | ZN-- |
| | mult.16s.r | Multiply, 16-bit signed, register | ZN-- |
| | mult.16u.i | Multiply, 16-bit unsigned, immediate | ZN-- |
| | mult.16u.r | Multiply, 16-bit unsigned, register | ZN-- |
| | mult.i | Multiply, immediate | ZNCV |
| | mult.r | Multiply, register | ZNCV |
| | mult.sh.r | Multiply, signed shift right, register | ZNCV |
| Divide | div.s.r | Divide, signed, register | ZN-V |
| | div.u.r | Divide, unsigned, register | ZNC- |

| | | | |
|---|---|---|---|
| Remainder | rem.s.r | Remainder, signed, register | ZN-V |
| | rem.u.r | Remained, unsigned, register | ZNC- |

### 5.3.7 Compare operations

| | 32-bit | 16-bit | 8-bit |
|---|---|---|---|
| High immediate | cmp.h | cmp.16.h | invalid |
| Immediate | cmp.i | cmp.16.i | cmp.8.i |
| Register | cmp.r | cmp.16.r | cmp.8.r |
| High immediate with carry | cmp.c.h | cmp.16c.h | invalid |
| Immediate with carry | cmp.c.i | cmp.16c.i | cmp.8c.i |
| Register with carry | cmp.c.r | cmp.16c.r | cmp.8c.r |
| High immediate, exchanged order | cmp.x.h | cmp.16x.h | invalid |
| Immediate, exchanged order | cmp.x.i | cmp.16x.i | cmp.8x.i |
| High immediate with carry, exchanged order | cmp.xc.h | cmp.16xc.h | invalid |
| Immediate with carry, exchanged order | cmp.xc.i | cmp.16xc.i | cmp.8xc.i |

All cmp instructions affect the ZNCV flags.

### 5.3.8 Shift and rotate

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Shift | shiftl.64.i | Shift left, 64-bit, immediate | ---- |
| | shiftl.64.r | Shift left, 64-bit, register | ---- |
| | shiftl.r | Shift left, register | ZNC- |
| | shiftr.64s.i | Shift right, 64-bit signed, immediate | ---- |
| | shiftr.64s.r | Shift right, 64-bit signed, register | ---- |
| | shiftr.s.r | Shift right, signed, register | ZNC- |
| | shiftr.u.r | Shift right, unsigned, register | ZNC- |
| Rotate | rlinm.s | Rotate left and mask, sign-extend | ZNC- |
| | rlinm.z | Rotate left and mask, zero-extend | ZNC- |
| | rotatel.r | Rotate left, register | ZNC- |

### 5.3.9 DSP instructions

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Flip | flip.16b.r | Flip and byte swap, 16-bit words | ZN-- |
| | flip.16x.r | Flip and bit swap, 16-bit words | ZN-- |
| | flip.8x.r | Flip and bit swap, in bytes | ZN-- |
| | flip.b.r | Flip and byte swap | ZN-- |

| | | | |
|---|---|---|---|
| | flip.r | Flip | ZN-- |
| | flip.x.r | Flip and bit swap | ZN-- |
| Absolute | abs.r | Absolute, register | ---- |
| Find most-significant bit | msbit.r | Most significant bit, register | ---- |
| 64-bit accumulate | acc.64.r | Accumulate, 64-bit, register | ---- |

### 5.3.10 Miscellaneous instructions

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| System instructions | brk | Break | ---- |
| | halt | Halt | ---- |
| | sleepnop | Sleep | ---- |
| | sleepsif | Sleep and allow SIF | ---- |
| | nop | No operation | ---- |
| | print.r | Print, register | ---- |
| Debug | sif | SIF | ---- |
| | movb2r | Move breakpoint to register | ZN-- |
| | movr2b | Move register to breakpoint | ---- |
| | ver | Read XAP3 version number | ZN-- |
| | lic | Read XAP3 licence number | ZN-- |
| Interrupt and exceptions | rtie | Return from interrupt or exception | ZNCV |
| | irqe | Interrupt enable | ---- |
| | irqd | Interrupt disable | ---- |
| | irqs | Interrupt status | ZN-- |
| | trap.i | Trap, immediate | ---- |
| | trap.r | Trap, register | ---- |
| Special registers | movs2r | Move special to register | ZN-- |
| | movr2s | Move register to special | ---- |

## 5.4 Alphabetical List of All Instructions

The 182 XAP3 instructions are listed alphabetically below. The "Min Size" column indicates whether a 16-bit encoding is available for the instruction.

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| abs.r | Rd, Rs | Absolute, register | 32 |
| acc.64.r | Rd, Rs | Accumulate, 64-bit, register | 32 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| add.c.r | Rd, Rs, Rt | Add with carry, register | 32 |
| add.h | Rd, Rs, #immediate | Add, high immediate | 32 |
| add.i | Rd, Rs, #immediate | Add, immediate | 16 |
| add.r | Rd, Rs, Rt | Add, register | 16 |
| and.h | Rd, Rs, #immediate | And, high immediate | 32 |
| and.i | Rd, Rs, #immediate | And, immediate | 32 |
| and.r | Rd, Rs, Rt | And, register | 16 |
| bcase.2 | Rs, #immediate | Case branch into 2 byte table | 32 |
| bcase.4 | Rs, #immediate | Case branch into 4 byte table | 32 |
| bcc | label | Branch if carry clear | 16 |
| bcs | label | Branch if carry set | 16 |
| beq | label | Branch if equal | 16 |
| bge.s | label | Branch if greater than or equal, signed | 16 |
| bge.u | label | Branch if greater than or equal, unsigned | 32 |
| bgt.s | label | Branch if greater than, signed | 32 |
| bgt.u | label | Branch if greater than, unsigned | 32 |
| ble.s | label | Branch if less than or equal, signed | 32 |
| ble.u | label | Branch if less than, unsigned | 32 |
| blt.s | label | Branch if less than, signed | 16 |
| blt.u | label | Branch if less than, unsigned | 32 |
| bmi | label | Branch if minus | 32 |
| bne | label | Branch if not equal | 16 |
| bpl | offset | Branch if plus | 32 |
| bra.a | address | Branch, absolute | 32 |
| bra.p<br>bra.p.2<br>bra.p.4 | label | Branch, PC-relative | 16 |
| bra.r | Ra | Branch, register | 16 |
| brk | | Break | 16 |
| bsr.a | address | Branch to subroutine, absolute | 32 |
| bsr.p | label | Branch to subroutine, PC-relative | 32 |
| bsr.r | Ra | Branch to subroutine, register | 16 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| bvc | label | Branch if overflow clear | 32 |
| bvs | label | Branch if overflow set | 32 |
| cmp.16.h | Rs, #immediate | Compare, 16-bit, high immediate | 32 |
| cmp.16.i | Rs, #immediate | Compare, 16-bit, immediate | 32 |
| cmp.16.r | Rs, Rt | Compare, 16-bit, register | 32 |
| cmp.16c.h | Rs, #immediate | Compare, 16-bit with carry, high immediate | 32 |
| cmp.16c.i | Rs, #immediate | Compare, 16-bit with carry, immediate | 32 |
| cmp.16c.r | Rs, Rt | Compare, 16-bit with carry, register | 32 |
| cmp.16x.h | Rs, #immediate | Compare, 16-bit, exchanged order, high immediate | 32 |
| cmp.16x.i | Rs, #immediate | Compare, 16-bit, exchanged order, immediate | 32 |
| cmp.16xc.h | Rs, #immediate | Compare, 16-bit with carry, exchanged order, high immediate | 32 |
| cmp.16xc.i | Rs, #immediate | Compare, 16-bit with carry, exchanged order, immediate | 32 |
| cmp.8.i | Rs, #immediate | Compare, immediate | 32 |
| cmp.8.r | Rs, Rt | Compare, register | 32 |
| cmp.8c.i | Rs, #immediate | Compare, 8-bit with carry, immediate | 32 |
| cmp.8c.r | Rs, Rt | Compare, 8-bit with carry, register | 32 |
| cmp.8x.i | Rs, #immediate | Compare, 8-bit, exchanged order, immediate | 32 |
| cmp.8xc.i | Rs, #immediate | Compare, 8-bit with carry, exchanged order, immediate | 32 |
| cmp.c.h | Rs, #immediate | Compare, with carry, high immediate | 32 |
| cmp.c.i | Rs, #immediate | Compare, with carry, immediate | 32 |
| cmp.c.r | Rs, Rt | Compare, with carry, register | 32 |
| cmp.h | Rs, #immediate | Compare, high immediate | 32 |
| cmp.i | Rs, #immediate | Compare, immediate | 16 |
| cmp.r | Rs, Rt | Compare, register | 16 |
| cmp.x.h | Rs, #immediate | Compare, exchanged order, high immediate | 32 |
| cmp.x.i | Rs, #immediate | Compare, exchanged order, immediate | 32 |
| cmp.xc.h | Rs, #immediate | Compare, with carry, exchanged order, | 32 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| | | high immediate | |
| cmp.xc.i | Rs, #immediate | Compare, with carry, exchanged order, immediate | 32 |
| div.s.r | Rd, Rs, Rt | Divide, signed, register | 16 |
| div.u.r | Rd, Rs, Rt | Divide, unsigned, register | 16 |
| flip.16b.r | Rd, Rs, Rt, #sel | Flip and byte swap, 16-bit words | 32 |
| flip.16x.r | Rd, Rs, Rt, # sel | Flip and bit swap, 16-bit words | 32 |
| flip.8x.r | Rd, Rs, Rt, # sel | Flip and bit swap, bytes | 32 |
| flip.b.r | Rd, Rs, Rt, # sel | Flip and byte swap | 32 |
| flip.r | Rd, Rs, Rt, # sel | Flip | 32 |
| flip.x.r | Rd, Rs, Rt, # sel | Flip and bit swap | 32 |
| halt | | Halt | 32 |
| irqd | | Disable interrupts | 32 |
| irqe | | Enable interrupts | 32 |
| irqs | Rd | Interrupt status | 32 |
| ld.16s.i | Rd, @(offset, Ra) | Load, 16-bit, sign-extend, displacement | 32 |
| ld.16s.r | Rd, @(Rx, Ra) | Load, 16-bit, sign-extend, indexed | 32 |
| ld.16z.i | Rd, @(offset, Ra) | Load, 16-bit, zero-extend, displacement | 16 |
| ld.16z.r | Rd, @(Rx, Ra) | Load, 16-bit, zero-extend, indexed | 32 |
| ld.8s.i | Rd, @(offset, Ra) | Load 8-bit, sign-extend, displacement | 32 |
| ld.8s.r | Rd, @(Rx, Ra) | Load, 8-bit, sign-extend, indexed | 32 |
| ld.8z.i | Rd, @(offset, Ra) | Load, 8-bit, zero-extend, displacement | 16 |
| ld.8z.r | Rd, @(Rx, Ra) | Load, 8-bit, zero-extend, indexed | 32 |
| ld.i | Rd, @(offset, Ra) | Load, displacement | 16 |
| ld.r | Rd, @(Rx, Ra) | Load, indexed | 32 |
| ldm | RegList, @(offset, Ra) | Load multiple | 16 |
| ldm.u | RegList, @(offset, Ra) | Load multiple, user | 32 |
| ldm.x | RegList, @(offset, Ra) | Load multiple, exchanged order | 32 |
| lic | Rd | Read XAP3 licence number | 32 |
| mov.g | Rd, #label | Move, GP-relative | 32 |
| mov.h | Rd, #immediate | Move, high immediate | 32 |
| mov.i | Rd, #immediate | Move, immediate | 16 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| mov.p | Rd, #label | Move, PC-relative | 32 |
| mov.r | Rd, Rt | Move, register | 16 |
| movb2r | Rd, BRs | Move breakpoint to register | 32 |
| movm | Rs1/#immediate1,<br>Rs2/#immediate2,<br>Rs3/#immediate3,<br>Rs4/#immediate4,<br>Rs5/#immediate5 | Move multiple | 16 |
| movm.x | Rs1/#immediate1,<br>Rs2/#immediate2,<br>Rs3/#immediate3,<br>Rs4/#immediate4,<br>Rs5/#immediate5 | Move multiple, exchanged order | 32 |
| movr2b | BRd, Rs | Move register to breakpoint | 32 |
| movr2s | SRd, Rs | Move register to special | 32 |
| movs2r | Rd, SRs | Move special to register | 32 |
| msbit.r | Rd, Rs | Most significant bit register | 32 |
| mult.16s.i | Rd, Rs, #immediate | Multiply, 16-bit signed, immediate | 32 |
| mult.16s.r | Rd, Rs, Rt | Multiply, 16-bit signed, register | 32 |
| mult.16u.i | Rd, Rs, #immediate | Multiply, 16-bit unsigned, immediate | 32 |
| mult.16u.r | Rd, Rs, Rt | Multiply, 16-bit unsigned, register | 32 |
| mult.i | Rd, Rs, #immediate | Multiply immediate | 32 |
| mult.r | Rd, Rs, Rt | Multiply, register | 16 |
| mult.sh.r | Rd, Rs, Rt, #immediate | Multiply, signed shift right, register | 32 |
| nop | | No operation | 16 |
| or.h | Rd, Rs, #immediate | Or, high immediate | 32 |
| or.i | Rd, Rs, #immediate | Or, immediate | 32 |
| or.r | Rd, Rs, Rt | Or, register | 16 |
| pop | RegList, #offset | Pop from stack | 16 |
| pop.ret | Rs/#immediate,<br>RegList,<br>#offset | Pop from stack and return | 16 |
| print.r | Rs | Print, register | 32 |
| push | RegList, #offset | Push to stack | 16 |
| push.r | RegList, #offset | Push to stack and registers | 16 |
| rem.s.r | Rd, Rs, Rt | Remainder, signed, register | 16 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| rem.u.r | Rd, Rs, Rt | Remainder, unsigned, register | 32 |
| rlinm.s | Rd, Rs, #shift, #highbit, #lowbit | Rotate left and mask, sign-extend | 16 |
| rlinm.z | Rd, Rs, #shift, #highbit, #lowbit | Rotate left and mask, zero-extend | 16 |
| rotatel.r | Rd, Rs, Rt | Rotate left, register | 32 |
| rtie | | Return from interrupt/exception | 32 |
| scc.i | Rd, Rs, #immediate | Set if carry clear, immediate | 32 |
| scc.r | Rd, Rs, Rt | Set if carry clear, register | 32 |
| scs.i | Rd, Rs, #immediate | Set if carry set, immediate | 32 |
| scs.r | Rd, Rs, Rt | Set if carry set, register | 32 |
| seq.i | Rd, Rs, #immediate | Set if equal, immediate | 16 |
| seq.r | Rd, Rs, Rt | Set if equal, register | 32 |
| sge.s.i | Rd, Rs, #immediate | Set if greater than or equal, signed, immediate | 32 |
| sge.s.r | Rd, Rs, Rt | Set if greater than or equal, signed, register | 32 |
| sge.u.i | Rd, Rs, #immediate | Set if greater than or equal, unsigned, immediate | 32 |
| sge.u.r | Rd, Rs, Rt | Set if greater than or equal, unsigned, register | 32 |
| sgt.s.i | Rd, Rs, #immediate | Set if greater than, signed, immediate | 32 |
| sgt.s.r | Rd, Rs, Rt | Set if greater than, signed, register | 32 |
| sgt.u.i | Rd, Rs, #immediate | Set if greater than, unsigned, immediate | 32 |
| sgt.u.r | Rd, Rs, Rt | Set if greater than, unsigned, register | 32 |
| shiftl.64.i | Rd, #immediate | Shift left, 64-bit, immediate | 32 |
| shiftl.64.r | Rd, Rt | Shift left, 64-bit, register | 32 |
| shiftl.r | Rd, Rs, Rt | Shift left, register | 16 |
| shiftr.64s.i | Rd, #immediate | Shift right, 64-bit signed, immediate | 32 |
| shiftr.64s.r | Rd, Rt | Shift right, 64-bit signed, register | 32 |
| shiftr.s.r | Rd, Rs | Shift right, signed, register | 16 |
| shiftr.u.r | Rd, Rs, Rt | Shift right, unsigned, register | 16 |
| sif | | SIF | 16 |
| sle.s.i | Rd, Rs, #immediate | Set if less than or equal, signed, immediate | 32 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| sle.s.r | Rd, Rs, Rt | Set if less than or equal, signed, register | 32 |
| sle.u.i | Rd, Rs, #immediate | Set if less than or equal, unsigned, immediate | 32 |
| sle.u.r | Rd, Rs, Rt | Set if less than, unsigned, register | 32 |
| sleepnop | | Sleep | 32 |
| sleepsif | | Sleep and allow SIF | 32 |
| slt.s.i | Rd, Rs, #immediate | Set if less than, signed, immediate | 32 |
| slt.s.r | Rd, Rs, Rt | Set if less than, signed, register | 32 |
| slt.u.i | Rd, Rs, #immediate | Set if less than, unsigned, immediate | 32 |
| slt.u.r | Rd, Rs, Rt | Set if less than, unsigned, register | 32 |
| smi.i | Rd, Rs, #immediate | Set if minus, immediate | 32 |
| smi.r | Rd, Rs, Rt | Set if minus, register | 32 |
| sne.i | Rd, Rs, #immediate | Set if not equal, immediate | 16 |
| sne.r | Rd, Rs, Rt | Set if not equal, register | 32 |
| spl.i | Rd, Rs, #immediate | Set if plus, immediate | 32 |
| spl.r | Rd, Rs, Rt | Set if plus, register | 32 |
| st.16.i | Rs, @(offset, Ra) | Store, 16-bit, displacement | 16 |
| st.16.r | Rs, @(Rx, Ra) | Store, 16-bit, indexed | 32 |
| st.8.i | Rs, @(offset, Ra) | Store, 8-bit, displacement | 16 |
| st.8.r | Rs, @(Rx, Ra) | Store, 8-bit, indexed | 32 |
| st.i | Rs, @(offset, Ra) | Store, displacement | 16 |
| st.r | Rs, @(Rx, Ra) | Store, indexed | 32 |
| stm | RegList, @(offset, Ra) | Store multiple | 16 |
| stm.u | RegList, @(offset, Ra) | Store multiple, user | 32 |
| stm.x | RegList, @(offset, Ra) | Store multiple, exchanged order | 32 |
| sub.c.r | Rd, Rs, Rt | Subtract, with carry, register | 32 |
| sub.r | Rd, Rs, Rt | Subtract, register | 16 |
| sub.x.h | Rd, Rs, #immediate | Subtract, exchanged order, high immediate | 32 |
| sub.x.i | Rd, Rs, #immediate | Subtract, exchanged order, immediate | 32 |
| svc.i | Rd, Rs, #immediate | Set if overflow clear, immediate | 32 |
| svc.r | Rd, Rs, Rt | Set if overflow clear, register | 32 |

| Mnemonic | Operands | Operation | Min Size |
|---|---|---|---|
| svs.i | Rd, Rs, #immediate | Set if overflow set, immediate | 32 |
| svs.r | Rd, Rs, Rt | Set if overflow set, register | 32 |
| swap.r | Rd, @(Rx, Ra) | Swap register with memory | 32 |
| trap.i | #immediate | Trap, immediate | 32 |
| trap.r | Rs | Trap, register | 32 |
| ver | Rd | Read XAP3 version number | 32 |
| xor.h | Rd, Rs, #immediate | Exclusive-or, high immediate | 32 |
| xor.i | Rd, Rs, #immediate | Exclusive-or, immediate | 16 |
| xor.r | Rd, Rs, Rt | Exclusive-or, register | 16 |

## 5.5 Privileged Instructions

The following instructions are not permitted in User Mode:

`rtie`, `ldm.u`, `stm.u`, `movr2s`, `movs2r`, `movr2b`, `movb2r`, `irqe`, `irqd`, `sleepsif`, `sleepnop`, `halt`, `sif`

These instructions either provide access to the FLAGS register (thereby allowing them to change the processor mode), or otherwise are capable of affecting the operation of the XAP3 core.

## 5.6 Aliased Instructions

The assembler translates a small number of instructions into other, equivalent instuctions. These are:

| Original instruction | Translated into |
|---|---|
| mov.h Rd, #immediate | or.h Rd, %r0, #immediate |
| mov.i Rd, #immediate | or.i Rd, %r0, #immediate |
| mov.r Rd, Rt | or.r Rd, %r0, Rt |
| blt.u label | bcs label |
| bge.u label | bcc label |
| slt.u.i Rd, Rs, #immediate | scs.i Rd, Rs, #immediate |
| sge.u.i Rd, Rs, #immediate | scc.i Rd, Rs, #immediate |
| slt.u.r Rd, Rs, Rt | scs.r Rd, Rs, Rt |
| sge.u.r | scc.r Rd, Rs, Rt |

The instructions are disassembled into the translated instructions by xIDE.

## 5.7 Register Naming in Instructions

Register operands are named as follows:

| Register symbol | Description |
|---|---|
| Rd | Destination register. |
| Rs | Primary source register. |
| Rt | Secondary source register. |
| Ra | Register containing a base address. This is interpreted as a byte address. |
| Rx | Register containing an index. This is interpreted according to the width of the instruction. Rx is used in the indexed addressing mode versions of the load and store instructions, and the `swap.r` instruction. |

## 5.8 Register Specification Fields

### *4-bit register fields*

Registers are generally specified with a 4-bit field. The mapping from the 4-bit field to the register set is as follows:

| 4-bit R field | register |
|---|---|
| 0000 | R0 |
| 0001 | R1 |
| 0010 | R2 |
| 0011 | R3 |
| 0100 | R4 |
| 0101 | R5 |
| 0110 | R6 |
| 0111 | R7 |

| 4-bit R field | register |
|---|---|
| 1000 | R8 |
| 1001 | R9 |
| 1010 | R10 |
| 1011 | R11 |
| 1100 | R12 |
| 1101 | R13 |
| 1110 | R14 |
| 1111 | R15 |

### *3-bit register fields*

Many of the 16-bit instruction encodings use a 3-bit field to encode a register. The mapping from the 3 bit field to the register set depends on the instruction. There are five mappings, called 'a' to 'e':

| | R?3a | R?3b | R?3c | R?3d | R?3e |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| 000 | R8 | R15 | R15 | R0 | R8 |
| 001 | R1 | R1 | R1 | R1 | R1 |
| 010 | R2 | R2 | R2 | R2 | R2 |
| 011 | R3 | R3 | R3 | R3 | R3 |
| 100 | R4 | R4 | R8 | R4 | R4 |
| 101 | R9 | R5 | R9 | R5 | R5 |
| 110 | R6 | R6 | R6 | R6 | R6 |
| 111 | R7 | R7 | R7 | R7 | R7 |

### 2-bit register fields

In some of the 16-bit encodings, registers are specified with a 2-bit field. This is mapped to the register set as follows:

| R?2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

### Special register fields

The mapping from a 4-bit special register field to the special registers is:

| SRx | register |
|---|---|
| 0000 | `%flags` |
| 0001 | `%sflags_s` |
| 0010 | `%sflags_i` |
| 0011 | `%sflags_n` |
| 0100 | `%gp_u` |
| 0101 | `%gp_s` |
| 0110 | `%gp_i` |
| 0111 | `%ivtb` |
| 1000 | `%brke` |
| 1001 to 1111 | reserved |

### Breakpoint register fields

The mapping from a 4-bit breakpoint register field to the breakpoint registers is:

| BRx | register |
|---|---|

| BRx | register |
|---|---|
| 0000 | %brk0 |
| etc | |
| 0111 | %brk7 |
| 1000 | %brk4count |
| 1001 | %brk5count |
| 1010 | %brk6count |
| 1011 | %brk7count |
| 1100 | %brk6mask |
| 1101 | %brk7mask |
| 1110 | %brk6data |
| 1111 | %brk7data |

# 6 Instruction Set Reference

The following pages describe each instruction in detail. Instructions are ordered alphabetically.

# abs.r — Absolute, register

| | | |
|---|---|---|
| **Instruction** | | `abs.r Rd, Rs` |
| **Description** | | Absolute value of a register |
| **Flags** | **Z** | Unchanged |
| | **N** | Unchanged |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = ( Rs > 0 ) ? Rs : -Rs` |
| **Usage Notes** | | - |
| **Examples** | | `abs.r %r1, %r2` |

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3 0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# acc.64.r — Accumulate, 64-bit, register

**Instruction** `acc.64.r Rd, Rs`

**Description** Accumulate 32-bit Rs into one of two 64-bit accumulators

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = Rd + Rs`

**Usage Notes** Rd is either R8 or R10. Other registers used as Rd throw an `UnknownInstruction` exception.

Rs can be any register.

**Examples**

```
acc.64.r %r8, %r1
acc.64.r %r10, %r2
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Rd[3:0] = 1000 means the 64-bit register {R9, R8}.

Rd[3:0] = 1010 means the 64-bit register {R11, R10}.

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# add.c.r — Add with carry, register

**Instruction** `add.c.r Rd, Rs, Rt`

**Description** Add two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| **V** | Set if the result of the signed operation is not correct; cleared otherwise |

**Operation** `Rd = Rs + Rt + FLAGS.C`

**Usage Notes** `add.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

**Examples**

```
add.c.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Rt[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# add.h — Add, high immediate

**Instruction** `add.h Rd, Rs, #immediate`

**Description** Add register and a high immediate

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared if the result is non-zero |
| N | Set if the result is negative; cleared if the result is positive |
| C | Set if the result of the unsigned operation is not correct; cleared otherwise |
| V | Set if the result of the signed operation is not correct; cleared otherwise |

**Operation** `Rd = Rs + #immediate`

**Usage Notes** Refer to section "h instructions - "h" encoding" for details of immediate encoding.

**Examples**

```
add.h %r1, %r2, #0x12340000
add.h %r1, %r2, #0x4321FFFF
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | #immediate[31:15]h | | | | 1 0 0 0 | 0 0 0 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# add.i — Add, immediate

**Instruction** `add.i Rd, Rs, #immediate`

**Description** Add a register and an immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| **V** | Set if the result of the signed operation is not correct; cleared otherwise |

**Operation** `Rd = Rs + #immediate`

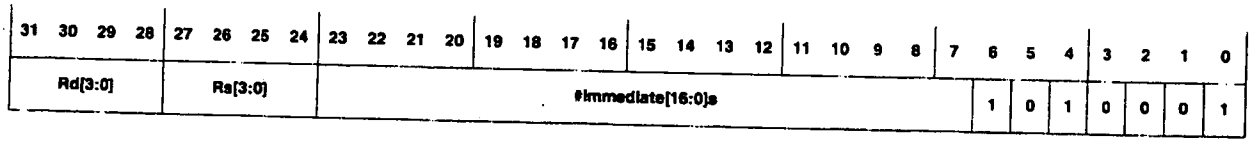
**Usage Notes** A number of commonly used combinations of registers and immediates are available as 16-bit encodings.

Refer to section "Instructions – "s" encoding" for details of immediate encoding.

**Examples**

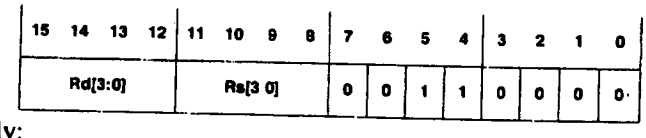
```
add.i %r1, %r2, #0x00001234
add.i %r1, %r2, #0xFFFF4321
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | #immediate[16:0]s | | | | 1 0 1 | 0 0 0 1 |

**16-bit Encoding – 1** `add.i Rd, Rs, #1` is encoded as:

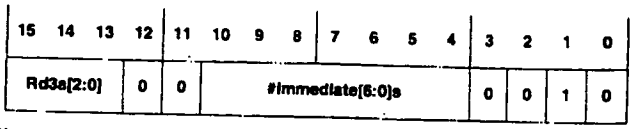


| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | Rs[3 0] | 0 0 1 1 | 0 0 0 0 |

The following rules apply:

- Rd=0 is reserved
- Rs=0 is reserved

**16-bit Encoding – 2** For immediates in the range –64, -63, …, -1, 1, 2, .. , 63, 64, the following encoding can be used:



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd3a[2:0] 0 | 0 | #immediate[6:0]s | 0 0 1 0 |

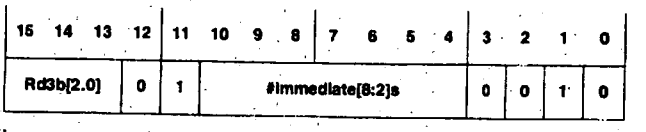
The following rules apply:

- A #immediate[6:0]s of zero is interpreted as +64. Hence, immediate values in the range –64, -63, …, -1, 1, 2, .. , 63, 64 can be expressed.
- Rd and Rs are the same register.
- The meaning of the 3-bit Rd3a field is:

| Rd3a | register |
|---|---|

| | |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

**16-bit Encoding – 3** `add.i Rd, %r15, #immediate` is encoded as:




The following rules apply:

- A #immediate[8:2]s of zero is interpreted as +256. Hence, immediate values in the range –256, -252, …, -4, 4, …, 252, 256 can be expressed.
- The meaning of the 3-bit Rd3b field is:

| Rd3b | register |
|---|---|
| 000 | R15 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

# add.r

## Add, register

**Instruction** `add.r Rd, Rs, Rt`

**Description** Add two registers

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| **V** | Set if the result of the signed operation is not correct; cleared otherwise |

**Operation** `Rd = Rs + Rt`

**Usage Notes** `add.r` can be used for signed or unsigned arithmetic.

**Examples**

```
add.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3.0] | 1 1 1 1 | 1 1 1 1 | 0 0 0 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | Rt[3:0] | 0 0 0 1 | 0 0 0 0 |

The following rules apply:

- Rd and Rs are the same register
- Rd=0 is reserved
- Rt=0 is reserved

# and.h — And, high immediate

**Instruction** `and.h Rd, Rs, #immediate`

**Description** Logical AND of register with high immediate value

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = Rs & #immediate`

**Usage Notes** Refer to section "h instructions – "h" encoding" for details of immediate encoding.

**Examples**

```
and.h %r1, %r2, #0x12340000
and.h %r1, %r2, #0x4321FFFF
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | #immediate[31:15]h | | | | 1 0 0 0 | 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# and.i — And, immediate

**Instruction** `and.i Rd, Rs, #immediate`

**Description** Logical AND of register with an immediate value

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero, cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = Rs & #immediate`

**Usage Notes** A number of specifc `and.i` operations can be implemented with the 16-bit `rlinm` instruction:

- `and.i Rd, Rd, #(2^imm+1-1)` can be implemented with `rlinm.z #0, #imm, #0`
- `and.i Rd, Rd, #2^imm` can be implemented with `rlinm.z #0, #imm, #imm`

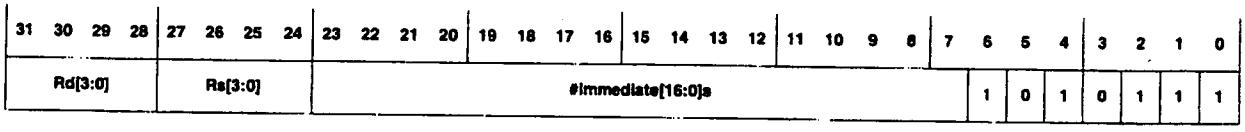
Refer to section "rlinstructions - "s" encoding" for details of immediate encoding.

**Examples**

```
and.i %r2, %r1, #0x00001234

and.i %r2, %r1, #0xFFFF1234
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# and.r

## And, register

**Instruction** `and.r Rd, Rs, Rt`

**Description** Logical AND of two registers

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = Rs & Rt`

**Usage Notes** -

**Examples**

```
and.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3.0] | Rs[3-0] | 1 1 1 1 | 1 1 1 1 | 0 0 1 1 | Rt[3-0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | Rt[3:0] | 0 1 0 0 | 0 0 0 0 |

The following rules apply:

- Rd and Rs are the same register
- Rd=0 is reserved
- Rt=0 is reserved

# bcase.2 — Case branch into 2 byte table

**Instruction**

```
bcase.2 Rs, #immediate
```

**Description**

Rs is compared against the immediate value. If Rs is less than the immediate value, the instruction generates a branch into a table following the instruction. The branch amount is Rs*2 bytes.

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( Rs < #immediate ) then
        PC = PC + 2 * ( Rs + 1 )
endif
```

**Usage Notes**

The bcase instructions are intended to be used to implement C switch-case statements. In a switch statement implementation, the bcase instructions test whether the switch variable is less than the number of case values and sets the PC to a position in a jump table according to the switch variable. The default case is not to take a branch. Therefore, the jump table starts with the default case.

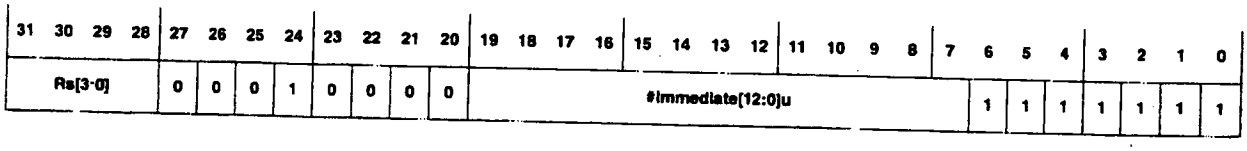
`bcase.4` is used with jump tables consisting of 4 byte branch instructions.

`bcase.2` is used with jump tables consisting of 2 byte branch instructions.

**Examples**

```
bcase.2 %r1, 3
bra.2 default
bra.2 r1_is_0
bra.2 r1_is_1
bra.2 r1_is 2
```

**32-bit Encoding**




**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# bcase.4 — Case branch into 4 byte table

**Instruction**

```
bcase.4 Rs, #immediate
```

**Description**

Rs is compared against the immediate value. If Rs is less than the immediate value, the instruction generates a branch into a table following the instruction. The branch amount is Rs*4 bytes.

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( Rs < #immediate ) then
        PC = PC + 4 * ( Rs + 1 )
endif
```

**Usage Notes**

The bcase instructions are intended to be used to implement C switch-case statements. In a switch statement implementation, the bcase instructions test whether the switch variable is less than the number of case values and sets the PC to a position in a jump table according to the switch variable. The default case is not to take a branch. Therefore, the jump table starts with the default case.

`bcase.4` is used with jump tables consisting of 4 byte branch instructions.

`bcase.2` is used with jump tables consisting of 2 byte branch instructions.

**Examples**

```
bcase.4 %r1, 3
bra.4 default
bra.4 r1_is_0
bra.4 r1_is_1
bra.4 r1_is_2
```

**32-bit Encoding**

| 31 30 29 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs[3:0] | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | #immediate[12:0]u | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# bcc — Branch if carry clear

**Instruction** `bcc label`

**Description** Branch if carry clear

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.C == 0 ) then
   PC = PC + offset
endif
```

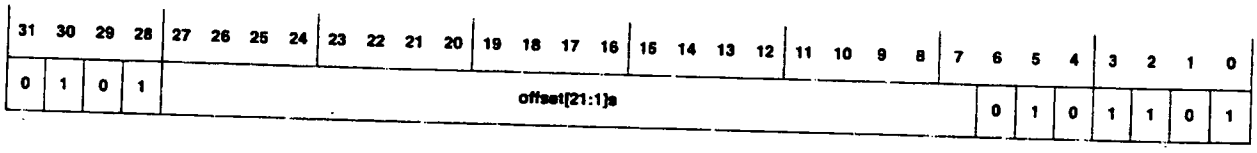
**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

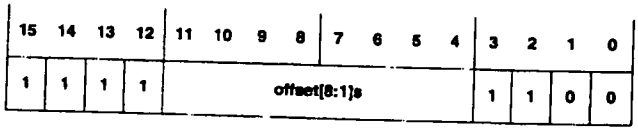
The `bge.u` (branch if greater than or equal, unsigned) instruction is translated into `bcc` by the assembler.

**Examples** `bcc label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -256, -254, ..., 252, 254, this 16-bit encoding can be used:



# bcs — Branch if carry set

**Instruction** `bcc label`

**Description** Branch if carry set

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

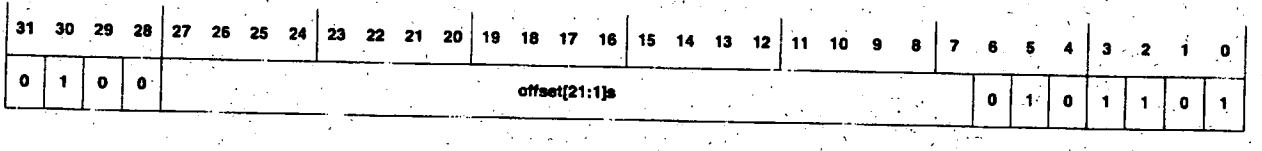
```
PC = next instruction
if ( FLAGS.C == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero

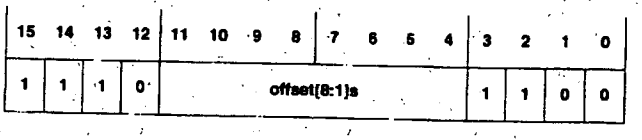
The `blt.u` (branch if less than, unsigned) instruction is translated into `bcs` by the assembler.

**Examples** `bcs label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -256, -254, ..., 252, 254, this 16-bit encoding can be used:



# beq — Branch if equal

**Instruction** `beq label`

**Description** Branch if equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.Z == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `beq label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | offset[21.1]s | | | | | | | | | | | | | | | | | | | | 0 | 1 | 0 | 1 | 1 | 0 | 1 | |

**16-bit Encoding** For offsets in the range -1024, -1022, ..., 1020, 1022, this 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | offset[10:1]s | | | | | | | | | | 1 | 1 | 0 | 0 |

# bge.s — Branch if greater than or equal, signed

**Instruction** `bge.s label`

**Description** Branch if greater than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

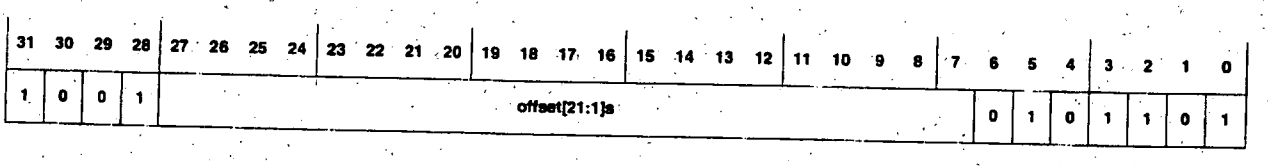
**Operation**

```
PC = next instruction
if ( FLAGS.N == FLAGS.V ) then
   PC = PC + offset
endif
```

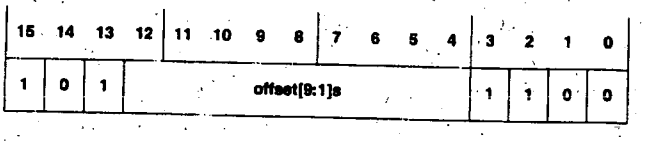
**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bge.s label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -512, -510, ..., 508, 510, this 16-bit encoding can be used:



# bge.u — Branch if greater than or equal, unsigned

**Instruction** `bge.u label`

**Description** Branch if greater than

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

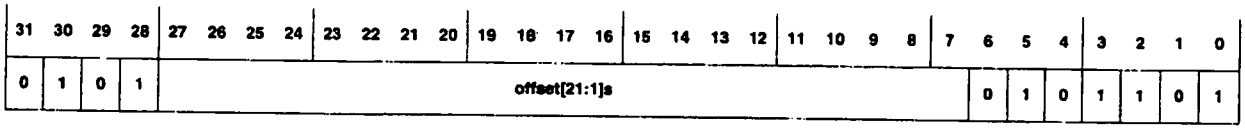
```
PC = next instruction
if ( FLAGS.C == 0 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

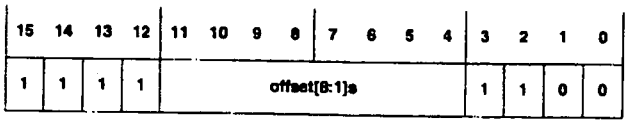
This instruction is translated into `bcc` by the assember. xIDE prints this instruction as `bcc`.

**Examples** `bge.u label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -256, -254, .., 252, 254, this 16-bit encoding can be used:



# bgt.s — Branch if greater than, signed

**Instruction** `bgt.s label`

**Description** Branch if greater than, signed

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

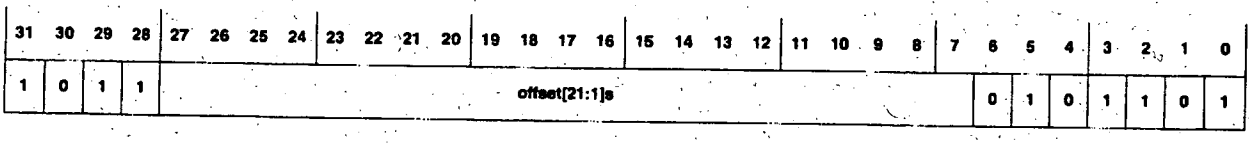
**Operation**

```
PC = next instruction
if ( ( FLAGS.N == FLAGS.V ) && ( FLAGS.Z == 0 ) ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bgt.s label`

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# bgt.u — Branch if greater than, unsigned

**Instruction** `bgt.u label`

**Description** Branch if greater than, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

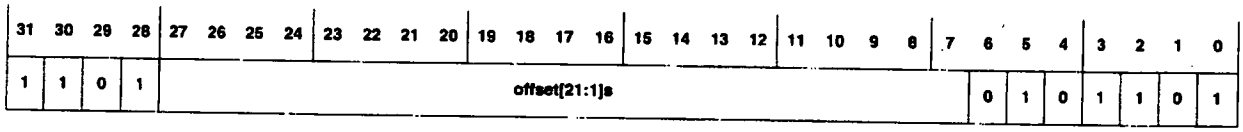
**Operation**

```
PC = next instruction
if ( ( FLAGS.C == 0 ) && ( FLAGS.Z == 0 ) ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bgt.u label`

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ble.s — Branch if less than or equal, signed

**Instruction** `ble.s label`

**Description** Branch if less than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

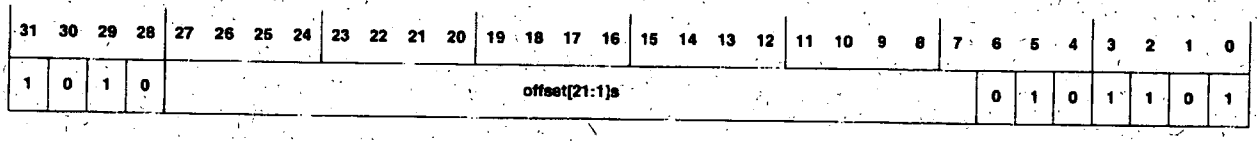
**Operation**

```
PC = next instruction
if ( ( FLAGS.N != FLAGS.V ) || ( FLAGS.Z == 1 ) ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `ble.s label`

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ble.u — Branch if less than or equal, unsigned

**Instruction** `ble.u label`

**Description** Branch if less than or equal, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

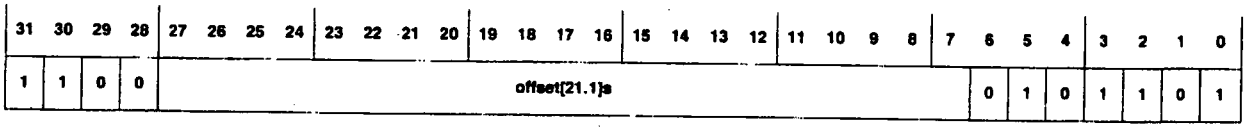
**Operation**

```
PC = next instruction
if ( ( FLAGS.C == 1 ) || ( FLAGS.Z == 1 ) ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `ble.u label`

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# blt.s — Branch if less than, signed

**Instruction** `blt.s label`

**Description** Branch if less than, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

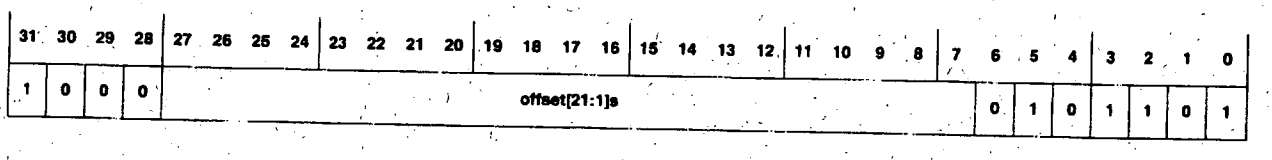
**Operation**

```
PC = next instruction
if ( FLAGS.N != FLAGS.V ) then
   PC = PC + offset
endif
```

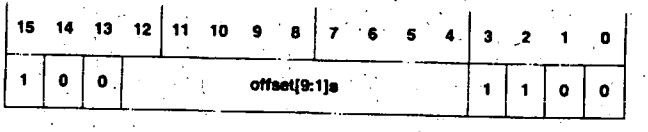
**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `blt.s label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -512, -510, ..., 508, 510, this 16-bit encoding can be used:



# blt.u — Branch if less than, unsigned

**Instruction** `blt.u offset`

**Description** Branch if less than

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.C == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

This instruction is translated into `bcs` by the assembler. xIDE prints this instruction as `bcs`.

**Examples** `blt.u label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | offset[21:1]s | | | | | | | | | | | | | | | | | | | | 0 | 1 | 0 | 1 | 1 | 0 | 1 | |

**16-bit Encoding** For offsets in the range -256, -254, ..., 252, 254, this 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | offset[8:1]s | | | | | | | | 1 | 1 | 0 | 0 |

# bmi — Branch if minus

**Instruction** `bmi label`

**Description** Branch if minus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

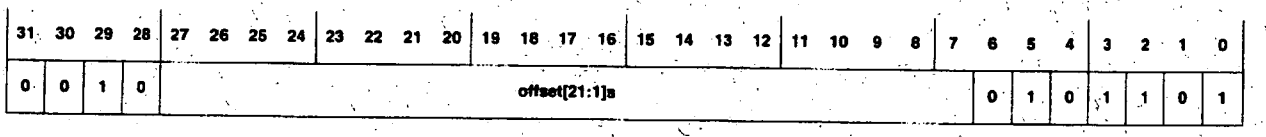
**Operation**

```
PC = next instruction
if ( FLAGS.N == 1 ) then
   PC = PC + offset
endif
```

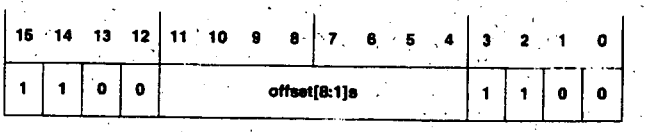
**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bmi label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -256, -254, .. , 252, 254 this 16-bit encoding can be used:



# bne — Branch if not equal

**Instruction** `bne label`

**Description** Branch if not equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

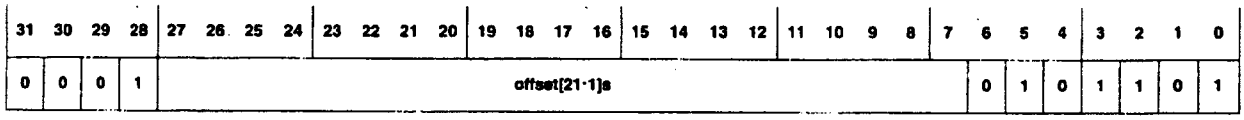
**Operation**

```
PC = next instruction
if ( FLAGS.Z == 0 ) then
   PC = PC + offset
endif
```

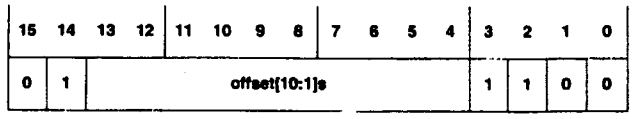
**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bne label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -1024, -1022, ..., 1020, 1022 , this 16-bit encoding can be used:



# bpl — Branch if plus

**Instruction** `bpl label`

**Description** Branch if plus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

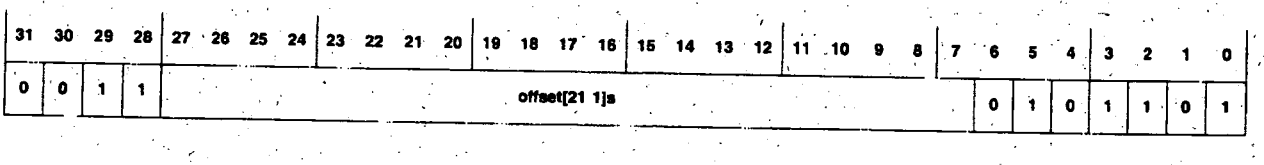
**Operation**

```
PC = next instruction
if ( FLAGS.N == 0 ) then
   PC = PC + offset
endif
```

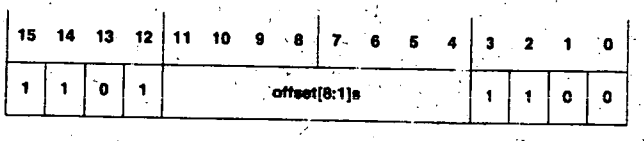
**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bpl label`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -256, -254,...., 252, 254 this 16-bit encoding can be used:



# bra.a — Branch, absolute

**Instruction** `bra.a address`

**Description** Unconditional branch to an absolute address

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `PC = address`

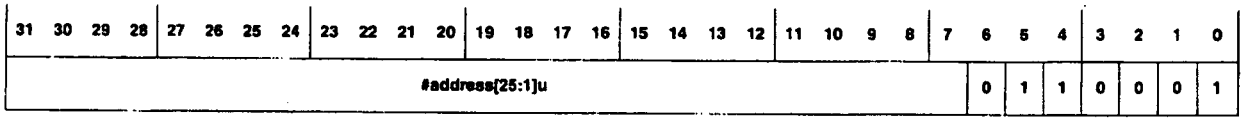
**Usage Notes** This instruction can branch anywhere in the first 64MByte of memory. To branch outside this area, use `bra.r`.

The address can be in the range 0 to 64MByte. To branch outside this range, the branch address can be formed in a register and `bra.r` used.

All instructions start on a 16-bit boundary, so the least significant bit of the address must be zero.

**Examples** `bra.a label`

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #address[25:1]u | | | | | | 0 1 1 | 0 0 0 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# bra.p, bra.p.2, bra.p.4 — Branch, PC-relative

**Instruction**

```
bra.p   label
bra.p.2 label
bra.p.4 label
```

**Description** Unconditional branch relative to the program counter

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = label
```

**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. All instructions start on a 16-bit boundary, so the least significant bit of the offset must be zero.

The 32-bit encoding can encode offsets in the range -32MByte to +32MByte. To branch outside this range, the branch address can be formed in a register and `bra.r` used.

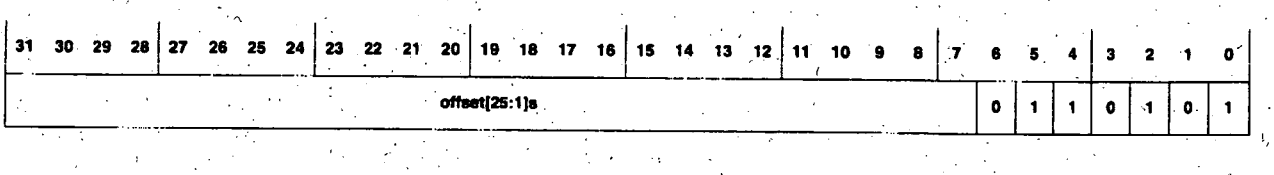
The 16-bit encoding can encode offsets in the range -2048 to +2046.

The assembler, by default, selects the 16-bit encoding for this instruction if the offset is in the range -2048 to +2046. Alternatively, the `bra.p.4` or `bra.p.2` instructions can be used to force the assembler to use the 32-bit or 16-bit encodings, respectively.

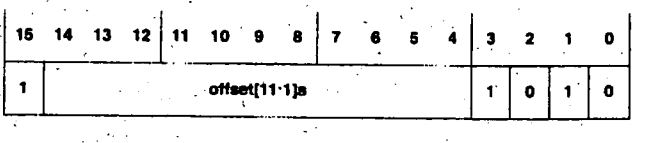
**Examples**

```
bra.p   label
bra.p.2 label
bra.p.4 label
```

**32-bit Encoding**




**16-bit Encoding** For offsets in the range -2048, -2046, ..., 2044, 2046, this 16-bit encoding can be used:



# bra.r — Branch, register

**Instruction** `bra.r Ra`

**Description** Unconditional branch to the address specified in the register Ra

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `PC = Ra`

**Usage Notes** This instruction can be used to return from simple subroutines (normally `bra.r %r14`).

This instruction throws an `IllegalCodeAddress` exception if bit 0 of Rs is not zero.

**Examples** `bra.r %r14`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Ra[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ra[3:0] | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# brk — Break

**Instruction** brk

**Description** Throw a break exception or halt the processor

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
if FLAGS.Mode == User && FLAGS.B == 1 then
      throw Break exception
else if debug enabled then
      halt
else
      nop
endif
```

**Usage Notes**

The brk instruction is used for debugging.

If the processor is in User Mode and the B bit of the FLAGS register is set, the Break exception is thrown. This allows an on-chip debugger or operating system to deal with the break.

Otherwise, if the processor is in debug mode the processor is stopped.

When debugging with the XAP3 simulator, the processor is always in debug mode. On the XAP3 emulator and real XAP3s, debug mode is controlled with the SIF interface.

The FLAGS register can be written with the movr2s instruction. This instruction is not permitted in User Mode.

The processor is restarted using the SIF interface or a reset.

**Examples**

```
brk
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# bsr.a — Branch to subroutine, absolute

**Instruction** `bsr.a address`

**Description** Unconditional branch to subroutine at an absolute address

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC  = next instruction
R14 = PC
PC  = address
```

**Usage Notes** The `bsr.a` instruction changes the program counter and saves the address of the instruction after the branch into the link register, R14.

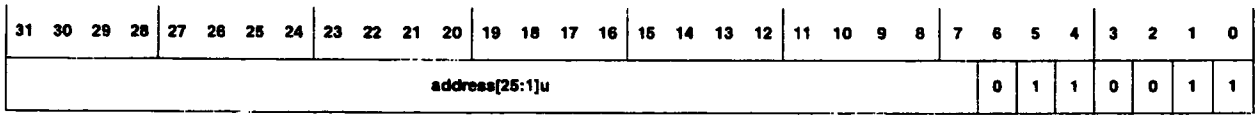
The address can be in the range 0 to 64MByte. To branch outside this range, the branch address can be formed in a register and `bsr.r` used.

All instructions start on a 16-bit boundary, so the least significant bit of the address must be zero.

**Examples**

```
bsr.a label
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# bsr.p — Branch to subroutine, PC-relative

**Instruction** `bsr.p label`

**Description** Unconditional branch to a subroutine at an address relative to the current PC

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC  = next instruction
R14 = PC
PC  = label
```

**Usage Notes** The `bsr.p` instruction changes the program counter and saves the address of the instruction after the branch into the link register, R14.

The assembler inserts the offset from the current PC to the label into the instruction. All instructions start on a 16-bit boundary, so the least significant bit of the offset must be zero.

The offset can take values in the range -32MByte to +32MByte. To branch outside this range, the branch address can be formed in a register and `bsr.r` used.

**Examples** `bsr.p label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| offset[25:1]s | | | | | | | | | | | | | | | | | | | | | | | | 0 | 1 | 1 | 0 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# bsr.r — Branch to subroutine, register

**Instruction** `bsr.r Ra`

**Description** Unconditional branch to a subroutine at an address contained in Ra

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC  = next instruction
R14 = PC
PC  = Ra
```

**Usage Notes** The `bsr.r` instruction changes the program counter and saves the address of the instruction after the branch into the link register, R14.

This instruction throws an `IllegalCodeAddress` exception if bit 0 of Rs is not zero.

**Examples**

```
bsr.r %r1
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Ra[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ra[3:0] | | | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# bvc — Branch if overflow clear

**Instruction** `bvc label`

**Description** Branch if overflow clear

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

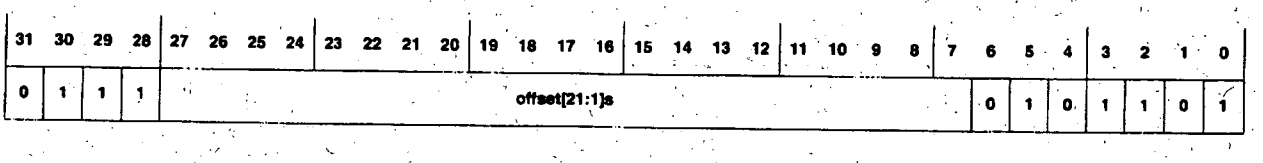
```
PC = next instruction
if ( FLAGS.V == 0 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples**

```
bvc label
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# bvs — Branch if overflow set

**Instruction** `bvs label`

**Description** Branch if overflow set

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

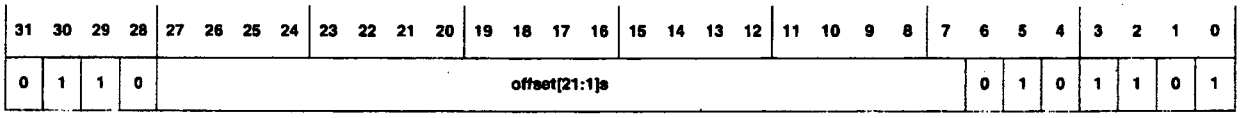
**Operation**

```
PC = next instruction
if ( FLAGS.V == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -2MByte to +2MByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bvs label`

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16.h — Compare, 16-bit, high immediate

**Instruction**

```
cmp.16.h Rs, #immediate
```

**Description**

A 16-bit compare of a register with a high immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[31:16] == #immediate[31:16]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation**

```
Rs[31:16] - #immediate[31:16]
```

**Usage Notes**

This is a 16-bit compare of a 16-bit high immediate with the upper 16 bits of a register. The low 16 bits of the immediate and register are ignored.

**Examples**

```
// The Z bit is set if R1 is 0x1234XXXX, where X means
// "don't care"
cmp.16.h %r1, 0x12340000

// The Z bit is set if R1 is 0x4321XXXX, where X means
// "don't care"
cmp.16.h %r1, 0x43210000
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | Rs[3:0] | | | | #immediate[31.16] | | | | | | | | | | | | | | | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# cmp.16.i Compare, 16-bit, immediate

**Instruction** `cmp.16.i Rs, #immediate`

**Description** A 16-bit compare of a register with an immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[15:0] == #immediate[15:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs[15:0] - #immediate[15:0]`

**Usage Notes** This is a 16-bit compare of a 16-bit immediate with the lower 16 bits of a register. The high 16 bits of the immediate and register are ignored.

The immediate can take values in the range -32768 to +32767.

**Examples**

```
// The Z bit is set if R1 is 0xXXXX1234, where X means
// "don't care"
cmp.16.i %r1, #0x00001234

// The Z bit is set if R1 is 0xXXXX4321, where X means
// "don't care"
cmp.16.i %r1, #0x00004321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | Rs[3:0] | | | | 0 | #immediate[15:0] | | | | | | | | | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16.r — Compare, 16-bit, register

**Instruction** `cmp.16.r Rs, Rt`

**Description** Compare the low 16 bits in two registers

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[15:0] == Rt[15:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs[15:0] - Rt[15:0]`

**Usage Notes** This is a 16-bit compare of the lower 16 bits from two registers. The high 16 bits of the two registers are ignored.

**Examples** `cmp.16.r %r1, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16c.h Compare, 16-bit with carry, high immediate

**Instruction** `cmp.16c.h Rs, #immediate`

**Description** A 16-bit compare of the register with a high immediate and the carry flag

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[31:16] == ( #immediate[31:16] + FLAGS.C )` and the Z flag was set prior to this instruction; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `( (Rs[31:16] - #immediate[31:16]) >> 16 ) - FLAGS.C`

**Usage Notes** This instruction can be used to implement a 32-bit compare of a register with a 32-bit immediate.

First, the lower 16-bits of the immediate are first compared with the register. Then the upper 32-bits are compared with the register and the carry flag.

For example, to compare R4 with 0x43211234:

```
cmp.16.i    %r1, #0x00001234
cmp.16c.h   %r1, #0x43210000
```

This instruction sets the Z flag only if the Z flag was already set and the high order bits of the register and immediate are equal.

**Examples**

```
cmp.16c.h %r1, #0x12340000

cmp.16c.h %r1, #0x43210000
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | Rs[3:0] | | | | #immediate[31:16] | | | | | | | | | | | | | | | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16c.i Compare, 16-bit with carry, immediate

**Instruction** `cmp.16c.i Rs, #immediate`

**Description** A 16-bit compare of the register with an immediate and the carry flag

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[15:0] == ( #immediate[15:0] + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs[15:0] - #immediate[15:0] - FLAGS.C`

**Usage Notes** This is a 16-bit compare of a 16-bit immediate with the lower 16 bits of a register and the carry flag. The high 16 bits of the immediate and register are ignored.

The immediate can take values in the range -32768 to +65535

**Examples**

```
cmp.16c.i %r1, #0x00001234
cmp.16c.i %r1, #0x00004321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | Rs[3:0] | | | | 0 | #immediate[15 0] | | | | | | | | | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16c.r Compare, 16-bit with carry, register

**Instruction** `cmp.16c.r Rs, Rt`

**Description** A 16-bit compare of two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[15:0] == ( Rt[15:0] + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs[15:0] - Rt[15:0] - FLAGS.C`

**Usage Notes** This is a 16-bit compare of the lower 16 bits from two registers and the carry bit. The high 16 bits of the two registers are ignored.

**Examples** `cmp.16c.r %r1, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16x.h Compare, 16-bit, exchange, high immediate

**Instruction** `cmp.16x.h Rs, #immediate`

**Description** A 16-bit compare of the register with a high immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[31:16] == #immediate[31:16]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `(#immediate[31:16] - Rs[31:16])>>16`

**Usage Notes** This is a 16-bit compare of the upper 16 bits of a register with a 16-bit high immediate. The low 16 bits of the register and immediate are ignored.

This is the same as `cmp.16.h` but with the operand order reversed.

**Examples**

```
cmp.16x.h %r1, #0x12340000
cmp.16x.h %r1, #0x43210000
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | Rs[3:0] | | | | #immediate[31.16] | | | | | | | | | | | | | | | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16x.i Compare, 16-bit, exchange, immediate

**Instruction** `cmp.16x.i Rs, #immediate`

**Description** A 16-bit compare of the register with an immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[15:0] == #immediate[15:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `#immediate[15:0] - Rs[15:0]`

**Usage Notes** This is a 16-bit compare of the lower 16 bits of a register with a 16-bit immediate. The high 16 bits of the register and immediate are ignored.

The immediate can take values in the range -32768 to +65535.

This is the same as `cmp.16.i` but with the operand order reversed.

**Examples**

```
cmp.16x.i %r1, #0x00001234
cmp.16x.i %r1, #0x00004321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | Rs[3:0] | | | | 0 | #immediate[15:0] | | | | | | | | | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16xc.h Compare, 16-bit, carry, exchange, high imm

**Instruction** `cmp.16xc.h Rs, #immediate`

**Description** A 16-bit compare of a register with a high immediate and the carry bit, and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[31:16] == ( #immediate[31:16] + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `( (#immediate[31:16] - Rs[31:16]) >> 16 ) - FLAGS.C`

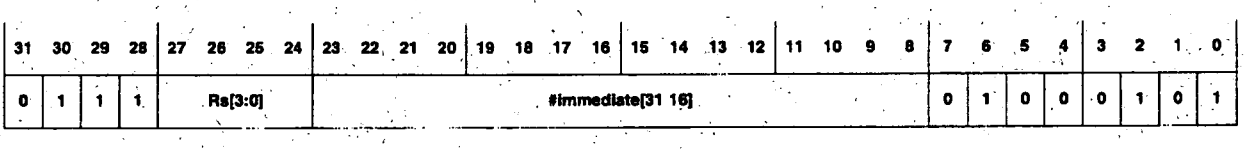
**Usage Notes** This is a 16-bit compare of a 16-bit high immediate with the high 16 bits of a register and the carry bit. The low 16 bits of the immediate and register are ignored.

This is the same as `cmp.16c.h` but with the operand order reversed.

**Examples**

```
cmp.16xc.h %r1, #0x12340000
cmp.16xc.h %r1, #0x43210000
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.16xc.i Compare, 16-bit, carry, exchange, immediate

**Instruction**

`cmp.16xc.i Rs, #immediate`

**Description**

A 16-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

**Flags**

**Z** Set if `Rs[15:0] == ( #immediate[15:0] + FLAGS.C )`; cleared otherwise

**N** Set if the result of the subtraction is negative; cleared if the result is positive

**C** Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise

**V** Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

**Operation**

`#immediate[15:0] - Rs[15:0] - FLAGS.C`

**Usage Notes**

This is a 16-bit compare of a 16-bit immediate with the low 16 bits of a register and the carry bit. The low 16 bits of the immediate and register are ignored.

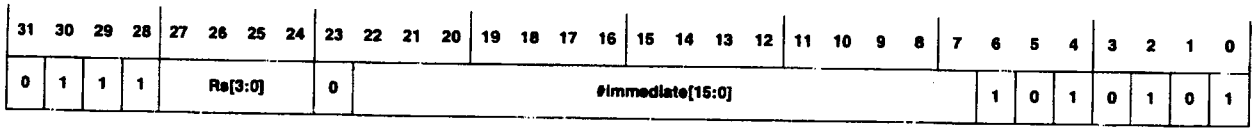
The immediate can take values in the range -32768 to +65535.

This is the same as `cmp.16c.i` but with the operand order reversed.

**Examples**

```
cmp.16xc.i %r1, #0x00001234
cmp.16xc.h %r1, #0x00004321
```

**32-bit Encoding**




**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# cmp.8.i — Compare, 8-bit, immediate

**Instruction** `cmp.8.i Rs, #immediate`

**Description** An 8-bit compare of the register with an immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[7:0] == #immediate[7:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation** `Rs[7:0] - #immediate[7:0]`

**Usage Notes** This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register. The high 24 bits of the immediate and register are ignored.

The immediate can take values in the range -127 to +255.

**Examples** `cmp.8.i %r1, #0x00000012`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Rs[3:0] | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[7:0] | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.8.r — Compare, 8-bit, register

**Instruction** `cmp.8.r Rs, Rt`

**Description** Compare the low 8 bits in two registers

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[7:0] == Rt[7:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation** `Rs[7:0] - Rt[7:0]`

**Usage Notes** This is an 8-bit compare of the lower 8 bits from two registers. The high 24 bits of the two registers are ignored.

**Examples** `cmp.8.r %r1, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.8c.i — Compare, 8-bit with carry, immediate

**Instruction**

```
cmp.8c.i Rs, #immediate
```

**Description**

An 8-bit compare of the register with an immediate and the carry flag

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[7:0] == ( #immediate[7:0] + FLAGS.C )`, cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits, cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation**

```
Rs[7:0] - #immediate[7:0] - FLAGS.C
```

**Usage Notes**

This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 24 bits of the immediate and register are ignored.

The immediate can take values in the range -127 to +255.

**Examples**

```
cmp.8c.i %r1, #0x00000012
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | Rs[3 0] | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[7:0] | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# cmp.8c.r Compare, 8-bit with carry, register

**Instruction** `cmp.8c.r Rs, Rt`

**Description** An 8-bit compare of two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[7:0] == ( Rt[7:0] + FLAGS.C )`, cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation** `Rs[7:0] - Rt[7:0] - FLAGS.C`

**Usage Notes** This is an 8-bit compare of the lower 8 bits from two registers and the carry bit The high 24 bits of the two registers are ignored.

**Examples** `cmp.8c.r %r1, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.8x.i — Compare, 8-bit, exchange, immediate

**Instruction** `cmp.8x.i Rs, #immediate`

**Description** An 8-bit compare of the register with an immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[7:0] == #immediate[7:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation** `#immediate[7:0] - Rs[7:0]`

**Usage Notes** This is an 8-bit compare of the lower 8 bits of a register with an 8-bit immediate. The high 24 bits of the register and immediate are ignored.

The immediate can take values in the range -127 to +255.

This is the same as `cmp.8.i` but with the operand order reversed.

**Examples** `cmp.8x.i %r1, #0x00000012`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | Rs[3:0] | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[7:0] | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.8xc.i Compare, 8-bit carry, exchange, immediate

**Instruction** `cmp.8xc.i Rs, #immediate`

**Description** An 8-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

**Flags**

**Z** Set if `Rs[7:0] == ( #immediate[7:0] + FLAGS.C )`; cleared otherwise

**N** Set if the result of the subtraction is negative; cleared if the result is positive

**C** Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise

**V** Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

**Operation** `#immediate[7:0] - Rs[7:0] - FLAGS.C`

**Usage Notes** This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 24 bits of the immediate and register are ignored.

The immediate can take values in the range -127 to +255.

This is the same as `cmp.8c.i` but with the operand order reversed.

**Examples** `cmp.8xc.i %r1, #0x00000012`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | Rs[3:0] | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[7:0] | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.c.h — Compare, with carry, high immediate

**Instruction** `cmp.c.h Rs, #immediate`

**Description** A 32-bit compare of the register with a high immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == ( #immediate + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative, cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `Rs - #immediate - FLAGS.C`

**Usage Notes** This is a 32-bit compare of a high immediate with a register and the carry bit.

Refer to section "h instructions - 'h' encoding" for details of immediate encoding.

**Examples**

```
cmp.c.h %r1, #0x12340000
cmp.c.h %r1, #0x4321FFFF
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | Rs[3.0] | | | | #immediate[31:15]h | | | | | | | | | | | | | | | | 1 | 0 | 0 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.c.i — Compare, with carry, immediate

**Instruction** `cmp.c.i Rs, #immediate`

**Description** A 32-bit compare of the register with an immediate and the carry flag

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == ( #immediate + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits, cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `Rs - #immediate - FLAGS.C`

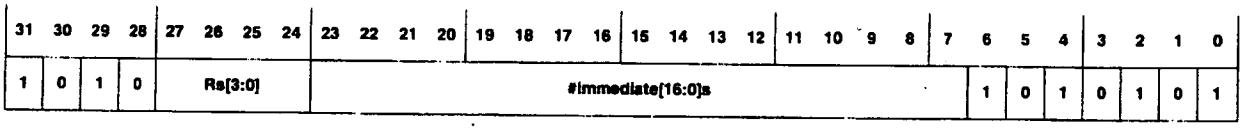
**Usage Notes** This is a 32-bit compare of an immediate with a register and the carry bit.

Refer to section "Instructions - "s" encoding" for details of immediate encoding.

**Examples**

```
cmp.c.i %r1, #0x00001234
cmp.c.i %r1, #0xFFFF4321
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.c.r — Compare, with carry, register

**Instruction** `cmp.c.r Rs, Rt`

**Description** A 32-bit compare of two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == ( Rt + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `Rs - Rt - FLAGS.C`

**Usage Notes** This is a 32-bit compare of two registers and the carry bit.

**Examples** `cmp.c.r %r1, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.h — Compare, high immediate

**Instruction** `cmp.h Rs, #immediate`

**Description** A 32-bit compare of the register with a high immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == #immediate`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `Rs - #immediate`

**Usage Notes** This is a 32-bit compare of a register with a high immediate.

Refer to section "h instructions - "h" encoding" for details of immediate encoding.

**Examples**

```
cmp.h %r1, 0x12340000
cmp.h %r1, 0x4321FFFF
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | Rs[3:0] | | | | #immediate[31:15]h | | | | | | | | | | | | | | | | 1 | 0 | 0 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.i — Compare, immediate

**Instruction** `cmp.i Rs, #immediate`

**Description** A 32-bit compare of the register with an immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == #immediate`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `Rs - #immediate`

**Usage Notes** This is a 32-bit compare of a register with an immediate.

Refer to section "Instructions -- 'S' encoding" for details of immediate encoding.

**Examples**

```
cmp.i %r1, #0x00001234

cmp.i %r1, #0xFFFF4321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | Rs[3:0] | | | | #immediate[16:0]s | | | | | | | | | | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** For immediates in the range 0 to 63, the following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs3a[2:0] | | | 1 | 0 | #immediate[6:0]u | | | | | | | 0 | 0 | 1 | 0 |

- The meaning of the 3-bit Rs3a field is:

| Rs3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

Cambridge Consultants

## cmp.r — Compare, register

**Instruction** `cmp.r Rs, Rt`

**Description** Compares two registers

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == Rt`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `Rs - Rt`

**Usage Notes** This is a 32-bit compare of two registers.

**Examples** `cmp.r %r1, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs[3:0] | | | | Rt[3:0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

# cmp.x.h — Compare, exchange, high immediate

**Instruction**

```
cmp.x.h Rs, #immediate
```

**Description**

A 32-bit compare of the register with a high immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == #immediate`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation**

```
#immediate - Rs
```

**Usage Notes**

This is a 32-bit compare of a high immediate with a register.

This is the same as `cmp.h` but with the operand order reversed.

Refer to section "h instructions - "h" encoding" for details of immediate encoding.

**Examples**

```
cmp.x.h %r1, 0x12340000

cmp.x.h %r1, 0x4321FFFF
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | Rs[3:0] | | | | #immediate[31.15]h | | | | | | | | | | | | | | | | 1 | 0 | 0 | 0 | 1 | 0 | 1 | |

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# cmp.x.i — Compare, exchange, immediate

**Instruction** `cmp.x.i Rs, #immediate`

**Description** A 32-bit compare of the register with an immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == #immediate`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits, cleared otherwise |

**Operation** `Rs - #immediate`

**Usage Notes** This is a 32-bit compare of an immediate with a register.

This is the same as `cmp.i` but with the operand order reversed.

Refer to section "Instructions - "s" encoding" for details of immediate encoding.

**Examples**

```
cmp.x.i %r1, #0x00001234
cmp.x.i %r1, #0xFFFF4321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | Rs[3:0] | | | | #Immediate[16:0]s | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 | 1 | | 0 | 1 | 0 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.xc.h Compare, with carry, exchange, high imm

**Instruction** `cmp.xc.h Rs, #immediate`

**Description** A 32-bit compare of a register with a high immediate and the carry bit, and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == ( #immediate + FLAGS.C )`, cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `#immediate - Rs - FLAGS.C`

**Usage Notes** This is a 32-bit compare of a high immediate with a register and the carry bit

This is the same as `cmp.c.h` but with the operand order reversed.

Refer to section "h instructions - "h" encoding" for details of immediate encoding.

**Examples**

```
cmp.xc.h %r1, #0x12340000

cmp.xc.h %r1, #0x4321FFFF
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | Rs[3.0] | | | | #immediate[31:15]h | | | | | | | | | | | | | | | | 1 | 0 | 0 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# cmp.xc.i Compare, with carry, exchange, immediate

**Instruction** `cmp.xc.i Rs, #immediate`

**Description** A 32-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == ( #immediate + FLAGS.C )`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 32 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise |

**Operation** `#immediate - Rs - FLAGS.C`

**Usage Notes** This is a 32-bit compare of an immediate with a register and the carry bit.

This is the same as `cmp.c.i` but with the operand order reversed.

Refer to section "i instructions – "s" encoding" for details of immediate encoding.

**Examples**

```
cmp.xc.i %r1, #0x00001234

cmp.xc.i %r1, #0xFFFF4321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | Rs[3:0] | | | | #immediate[16:0]s | | | | | | | | | | | | | | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# div.s.r — Divide, signed, register

**Instruction** `div.s.r Rd, Rs, Rt`

**Description** Signed 32-bit integer divide

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = Rs / Rt`

**Usage Notes** `div.s.r` truncates towards zero.

Attempts to divide by zero operate as follows:

- In User Mode, the processor throws a `DivideByZero` exception.
- In one of the privileged modes, the division does not take place and the destination register remains unchanged.

The division 0x80000000 / -1 gives a result of 0x80000000 and sets the overflow flag.

**Examples** `div.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 0 1 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# div.u.r — Divide, unsigned, register

**Instruction** `div.u.r Rd, Rs, Rt`

**Description** Unsigned 32-bit integer divide

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation** `Rd = Rs / Rt`

**Usage Notes** Attempts to divide by zero (Rt=0) operate as follows:

- In User Mode, the processor throws a `DivideByZero` exception.
- In one of the privileged modes, the division does not take place and the destination register remains unchanged

**Examples**

```
div.u.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 | 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 | 1 1 | 1 1 1 1 | 1 0 1 1 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# flip.16b.r — Flip and byte swap, 16-bit words

**Instruction**

```
flip.16b.r Rd, Rs, Rt, #sel
```

**Description**

Swap bytes within two 16-bit integers in a 32-bit word

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative, cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

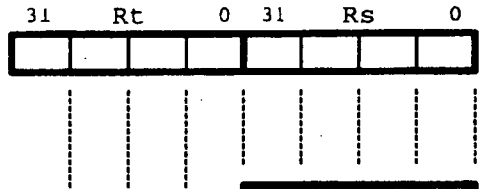
**Operation**

```
uint64 Rts;
uint32 Rsel;
Rts  = ( Rt << 32 ) | Rs
Rsel = ( Rts & ( 0xFFFFFFFF << (LB*8) ) ) >> (LB*8)
Rd   = byteswap2x16(Rsel)
```

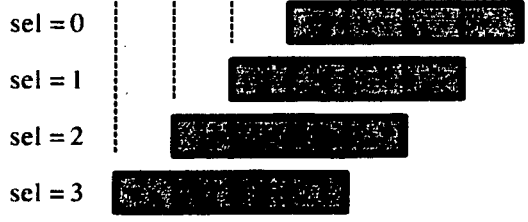
**Usage Notes**

`sel` is one of the values 0, 1 ,2, 3.

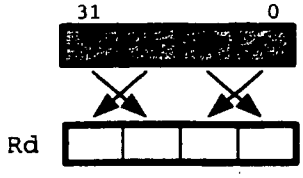
The flip instructions construct a 64-bit integer from Rt and Rs:




The 2-bit sel operand selects a 32-bit slice from the 64-bit integer:




Finally, the `flip.16b.r` instruction interprets the 32-bit integer as two 16-bit integers, and performs byte swapping on each




**Examples**

To convert the byte ordering of two 16-bit integers in R2 and place the result in R1:

```
flip.16b.r %r1, %r2, %r2, #0
```

To select the middle four bytes from {R8, R7}, do a 16-bit byte swap and put the result in R6:

```
flip.16b.r %r6, %r7, %r8, #2
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | 0 | 0 | 1 | 0 | 0 | #sel[1 0] | | 0 | 1 | 0 | 1 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# flip.16x.r — Flip and bit swap, 16-bit words

**Instruction**

```
flip.16x.r Rd, Rs, Rt, #sel
```

**Description**

Swap bits within two 16-bit integers in a 32-bit word

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

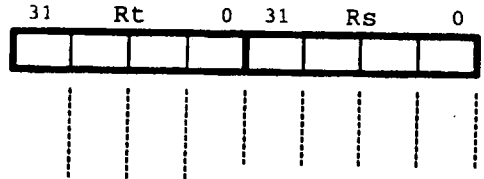
**Operation**

```
uint64 Rts;
uint32 Rsel;
Rts  = ( Rt << 32 ) | Rs
Rsel = ( Rts & ( 0xFFFFFFFF << (sel*8) ) ) >> (sel*8)
Rd   = bitswap2x16(Rsel)
```

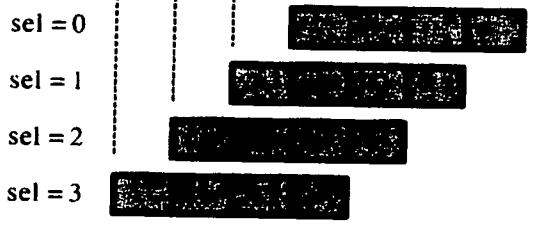
**Usage Notes**

`sel` is one of the values 0, 1 ,2, 3.

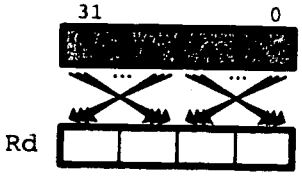
The flip instructions construct a 64-bit integer from Rt and Rs:




The 2-bit sel operand selects a 32-bit slice from the 64-bit integer:




Finally, the `flip.16x.r` instruction interprets the 32-bit integer as two 16-bit integers, and performs bit swapping within each:




**Examples**

To reverse the order of the bits in two 16-bit integers in R2 and place the result in R1:

```
flip.16b.r %r1, %r2, %r2, #0
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3 0] | | | | Rs[3:0] | | | | 0 | 0 | 0 | 1 | 0 | 0 | #sel[1:0] | | 0 | 0 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# flip.8x.r — Flip and bit swap, bytes

**Instruction** `flip.8x.r Rd, Rs, Rt, #sel`

**Description** Swap bits within four 8-bit integers in a 32-bit word

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

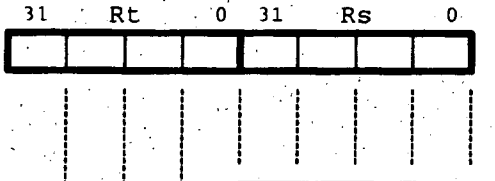
**Operation**

```
uint64 Rts;
uint32 Rsel;
Rts  = ( Rt << 32 ) | Rs
Rsel = ( Rts & ( 0xFFFFFFFF << (sel*8) ) ) >> (sel*8)
Rd   = bitswap4x8(Rsel)
```

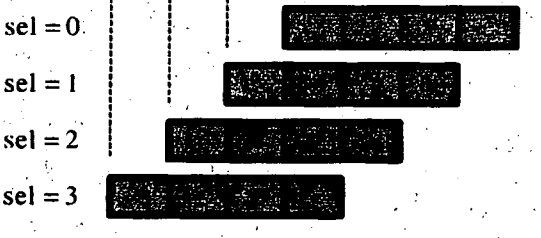
**Usage Notes**

`sel` is one of the values 0, 1 ,2, 3.

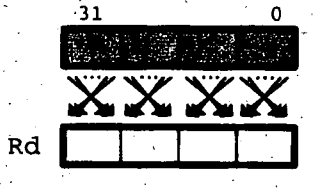
The flip instructions construct a 64-bit integer from Rt and Rs:




The 2-bit sel operand selects a 32-bit slice from the 64-bit integer:




Finally, the `flip.8x.r` instruction interprets the 32-bit integer as four8-bit characters, and performs bit swapping within each:




**Examples** To reverse the order of the bits in four 8-bit characters in R2 and place the result in R1:

```
flip.8x.r %r1, %r2, %r2, #0
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 0 | 0 | 0 | 1 | 0 | 0 | #sel[1:0] | | 0 | 0 | 1 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# flip.b.r — Flip and byte swap

**Instruction** `flip.b.r Rd, Rs, Rt, #sel`

**Description** Swap bytes within one 32-bit integer in a 32-bit word

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

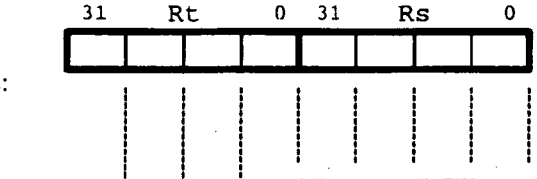
**Operation**

```
uint64 Rts;
uint32 Rsel;
Rts  = ( Rt << 32 ) | Rs
Rsel = ( Rts & ( 0xFFFFFFFF << (sel*8) ) ) >> (sel*8)
Rd   = byteswap1x32(Rsel)
```

**Usage Notes**

`sel` is one of the values 0, 1 ,2, 3.

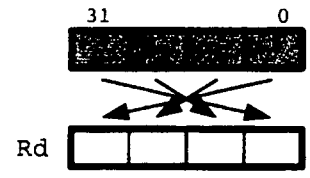
The flip instructions construct a 64-bit integer from Rt and Rs:




The 2-bit sel operand selects a 32-bit slice from the 64-bit integer:




Finally, the `flip.b.r` instruction performs byte swapping within the 32-bit slice:




**Examples** To reverse the order of the bytes in R2 and place the result in R1:

```
flip.b.r %r1, %r2, %r2, #0
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 0 0 1 | 0 0 #sel[1:0] | 0 1 0 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# flip.r



**Instruction** `flip.r Rd, Rs, Rt, #sel`

**Description** Select four bytes from a 64-bit word

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

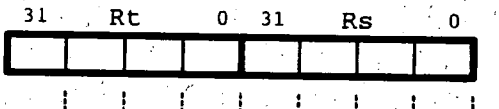
**Operation**

```
uint64 Rts;
uint32 Rsel;
Rts  = ( Rt << 32 ) | Rs
Rsel = ( Rts & ( 0xFFFFFFFF << (sel*8) ) ) >> (sel*8)
Rd   = Rsel
```

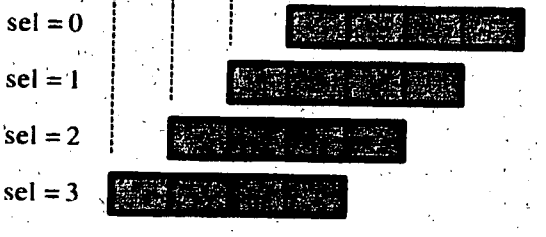
**Usage Notes**

`sel` is one of the values 0, 1 ,2, 3.

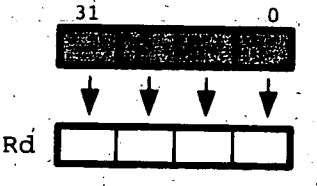
The flip instructions construct a 64-bit integer from Rt and Rs:




The 2-bit sel operand selects a 32-bit slice from the 64-bit integer:




The `flip.r` instruction does not perform any bit or byte swapping:




**Examples** To select the middle four bytes from {R10, R11} and place the result in R1:

```
flip.r %r1, %r11, %r10, #2
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 0 | 0 | 0 | 1 | 0 | 0 | #sel[1:0] | | 0 | 0 | 0 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# flip.x.r — Flip and bit swap

**Instruction** `flip.x.r Rd, Rs, Rt, #sel`

**Description** Swap bits within a 32-bit word

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

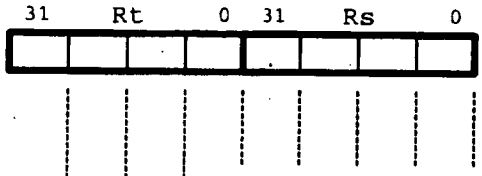
**Operation**

```
uint64 Rts;
uint32 Rsel;
Rts  = ( Rt << 32 ) | Rs
Rsel = ( Rts & ( 0xFFFFFFFF << (sel*8) ) ) >> (sel*8)
Rd   = bitswap1x32(Rsel)
```

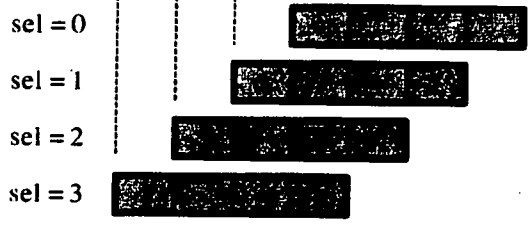
**Usage Notes**

`sel` is one of the values 0, 1 ,2, 3.

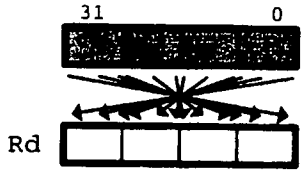
The flip instructions construct a 64-bit integer from Rt and Rs:




The 2-bit sel operand selects a 32-bit slice from the 64-bit integer:




Finally, the `flip.x.r` instruction reverses the order of all the bits:




**Examples** To select bytes the middle four bytes from {R8, R7}, do a 32-bit bitswap and copy the result to R6:

```
flip.x.r %r6, %r7, %r8, #0x3
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 0 | 0 | 0 | 1 | 0 | 0 | #sel[1.0] | | 0 | 0 | 0 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# halt

## Halt

**Instruction** `halt`

**Description** Stop the processor

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if (FLAGS.Mode == UserMode) then
   halt processor
else
   throw IllegalInstruction exception
endif
```

An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

The processor is restarted using the SIF interface or a reset.

**Examples**

```
halt
```

**32-bit Encoding**

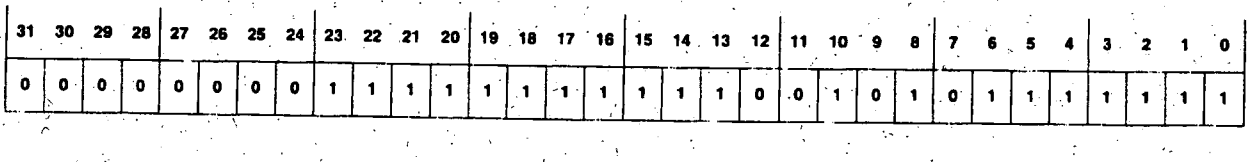
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# irqd — Disable interrupts

| | | |
|---|---|---|
| **Instruction** | | `irqd` |
| **Description** | | Disable all maskable interrupts |
| **Flags** | **Z** | Unchanged |
| | **N** | Unchanged |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `FLAGS.E = 0` |
| **Usage Notes** | | An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode. |
| **Examples** | | `irqd` |

**32-bit Encoding**



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# irqe — Enable interrupts

**Instruction** `irqe`

**Description** Enable maskable interrupts

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `FLAGS.E = 1`

**Usage Notes** An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

**Examples** `irqe`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# irqs — Interrupt status

| | | |
|---|---|---|
| **Instruction** | | `irqs Rd` |
| **Description** | | Read the interrupt enable bit |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set to zero |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = FLAGS.E ? 1 : 0` |
| **Usage Notes** | | This instruction reads the interrupt enable bit in the FLAGS register. |
| **Examples** | | `irqs %r1` |

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3-0] | | | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** — There is no 16-bit encoding for this instruction.

# ld.16s.i Load, 16-bit, sign-extend, displacement

**Instruction** `ld.16s.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra), sign-extend from 16 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (int32) (* (int16*) ( offset + Ra ) )
```

**Usage Notes**

Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535 bytes.

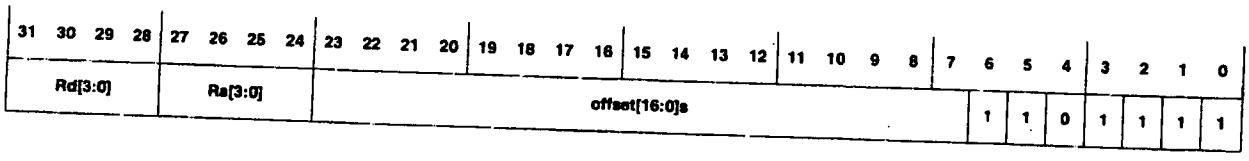
The memory value is sign-extended to 32 bits when written to Rd.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
ld.16s.i %r1, @(2, %r15)
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ld.16s.r — Load, 16-bit, sign-extend, indexed

**Instruction** `ld.16s.r Rd, @(Rx, Ra)`

**Description** Load Rd from address (2Rx + Ra), sign-extend from 16 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (int32) (* (int16*) ( 2*Rx + Ra ) )
```

**Usage Notes**

Ra is interpreted as a byte address. Rx is interpreted as a two-byte offset from Ra.

The memory value is sign-extended to 32 bits when written to Rd.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
ld.16s.r %r1, @(%r2, %r3)
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | 1 1 1 1 | 1 1 1 0 | 0 1 1 1 | Rx[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ld.16z.i — Load, 16-bit, zero-extend, displacement

**Instruction** `ld.16z.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra), zero-extend from 16 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (uint32) (* (int16*) ( offset + Ra ) )
```

**Usage Notes**

Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535 bytes.

The memory value is zero-extended to 32 bits when written to Rd.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `ld.16z.i %r1, @(2, %r15)`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | offset[16:0]s | | | | 1 1 0 | 0 0 1 1 |

**16-bit Encoding**

For offsets in the range 0, 2, ... 28, 30, the following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd3e[2:0] | Ra3c[2:0] 0 1 | offset[4:1]u | 0 1 0 0 |

The following rules apply:

- The meaning of the 3-bit Rd3e field is:

| Rd3e | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |

| | |
|---|---|
| 111 | R7 |

- The meaning of the 3-bit Ra3c field is:

| Ra3c | register |
|---|---|
| 000 | R15 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R8 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

# ld.16z.r Load, 16-bit, zero-extend, indexed

**Instruction** `ld.16z.r Rd, @(Rx, Ra)`

**Description** Load Rd from address (2Rx + Ra), zero-extend from 16 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (uint32) (* (int16*) ( 2*Rx + Ra ) )`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a two-byte offset from Ra.

The memory value is sign-extended to 32 bits when written to Rd.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU

**Examples** `ld.16z.r %r1, @(%r2, %r3)`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | 1 1 1 1 | 1 1 1 0 | 0 0 0 1 | Rx[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rx3a[2:0] | Ra3a[2:0] Rd2[1:0] | 1 0 0 1 | 0 0 0 0 |

The following rules apply:

- The meaning of the 3-bit Rx3a and Ra3a fields is:

| Rx3a, Ra3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

The meaning of the 2-bit Rd2 field is:

| Rd2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

# ld.8s.i — Load, 8-bit, sign-extend, displacement

**Instruction** `ld.8s.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra), sign-extend from 8 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (int32) (* (int8*) ( offset + Ra ) )
```

**Usage Notes**

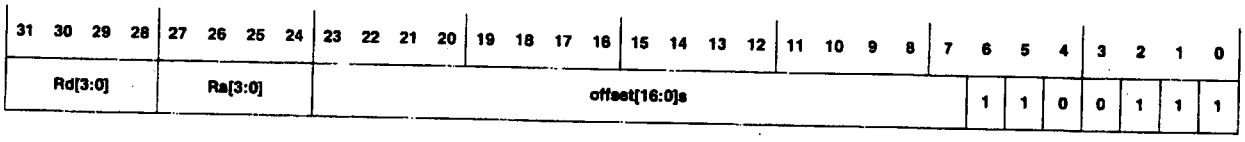
Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535 bytes.

The memory value is sign-extended to 32 bits when written to Rd.

An 8-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `ld.8s.i %r1, @(2, %r15)`

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ld.8s.r — Load, 8-bit, sign-extend, indexed

**Instruction** `ld.8s.r Rd, @(Rx, Ra)`

**Description** Load Rd from address (Rx + Ra), sign-extend from 8 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (int32) (* (int8*) ( Rx + Ra ) )
```

**Usage Notes**

Ra is interpreted as a byte address. Rx is interpreted as a byte offset from Ra.

The memory value is sign-extended to 32 bits when written to Rd.

An 8-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
ld.8s.r %r1, @(%r2, %r3)
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | Rx[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ld.8z.i Load, 8-bit, zero-extend, displacement

**Instruction** `ld.8z.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra), zero-extend from 8 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (uint32) (* (int8*) ( offset + Ra ) )
```

**Usage Notes** Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535 bytes.

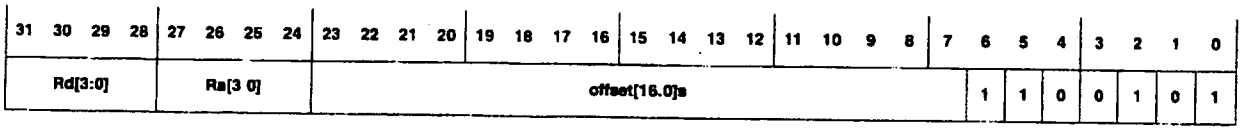
The memory value is zero-extended to 32 bits when written to Rd.

An 8-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

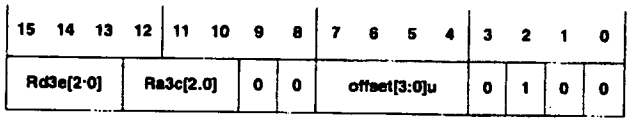
**Examples**

```
ld.8z.i %r1, @(2, %r15)
```

**32-bit Encoding**




**16-bit Encoding** For offsets in the range 0, 2, ... 28, 30, the following 16-bit encoding can be used:




The following rules apply:

- The meaning of the 3-bit Rd3e field is:

| Rd3e | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |

| | |
|---|---|
| 111 | R7 |

- The meaning of the 3-bit Ra3c field is:

| Ra3c | register |
|---|---|
| 000 | R15 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R8 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

# ld.8z.r — Load, 8-bit, zero-extend, indexed

**Instruction** `ld.8z.r Rd, @(Rx, Ra)`

**Description** Load Rd from address (Rx + Ra), zero-extend from 8 to 32 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (uint32) (* (int8*) ( Rx + Ra ) )`

**Usage Notes**

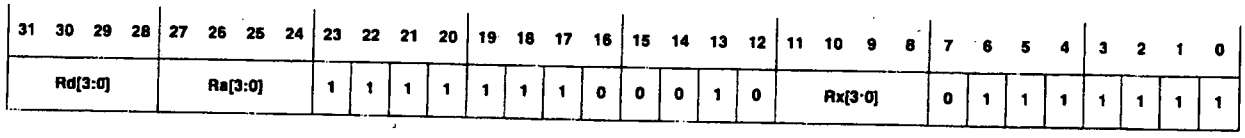
Ra is interpreted as a byte address. Rx is interpreted as a byte offset from Ra.

The memory value is sign-extended to 32 bits when written to Rd.

An 8-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

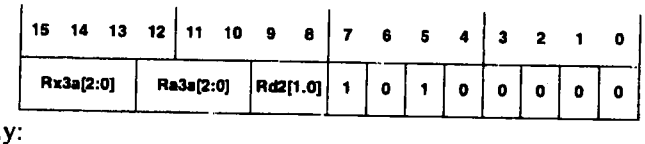
**Examples** `ld.8z.r %r1, @(%r2, %r3)`

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | 1 1 1 1 | 1 1 1 0 | 0 0 1 0 | Rx[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rx3a[2:0] | Ra3a[2:0] Rd2[1:0] | 1 0 1 0 | 0 0 0 0 |

The following rules apply:

- The meaning of the 3-bit Rx3a and Ra3a fields is:

| Rx3a, Ra3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

- The meaning of the 2-bit Rd2 field is:

| Rd2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

# ld.i — Load, displacement

**Instruction** `ld.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = * (int32*) ( offset + Ra )`

**Usage Notes** Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535 bytes.

A 32-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `ld.i %r1, @(2, %r15)`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | offset[16:0]s | | | | 1 1 0 | 0 0 0 1 |

**16-bit Encoding** For offsets in the range 0, 4, ..., 120, 124, the following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd3e[2:0] | Ra[3:0] | offset[6:2]u | 0 1 1 0 |

The following rules apply:

- The meaning of the 3-bit Rd3e field is:

| Rd3e | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

# ld.r — Load, indexed

**Instruction** `ld.r Rd, @(Rx, Ra)`

**Description** Load Rd from address (4Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

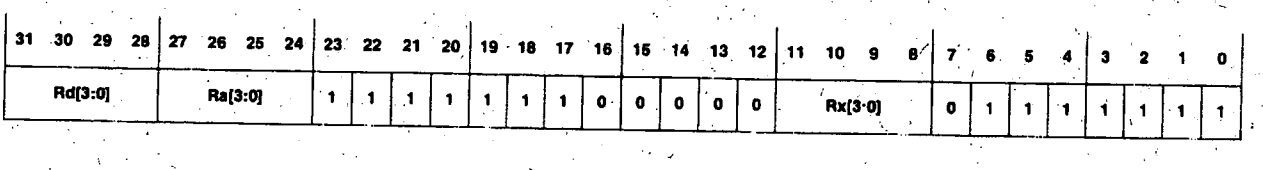
**Operation** `Rd = * (int32*) ( 4*Rx + Ra )`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a 4-byte offset from Ra.

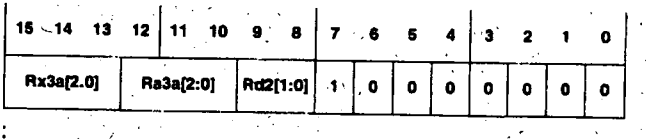
A 32-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `ld.r %r1, @(%r2, %r3)`

**32-bit Encoding**




**16-bit Encoding** The following 16-bit encoding can be used:




The following rules apply:

- The meaning of the 3-bit Rx3a and Ra3a fields is:

| Rx3a, Ra3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

- The meaning of the 2-bit Rd2 field is:

| Rd2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

# ldm — Load multiple

**Instruction** `ldm RegList, @(offset, Ra)`

**Description** Load multiple registers from memory

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the final load is zero; cleared otherwise |
| **N** | Set if the result of the final load is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

**Usage Notes**

`ldm` loads the registers being used by the current processor operating mode.

The RegList operand specifies which registers are to be loaded and can contain any of the registers in the range R0 to R15. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 120.

The instruction is atomic.

Higher numbered registers are loaded from higher memory addresses.

Registers are stored low register first. The flags are updated on the basis of the final load (highest register).

A `NullPointer` exception is thrown if any of the memory addresses are zero or within the MMU's defintion of null pointers, and the instruction terminates. Loads prior to the exception complete normally.

An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

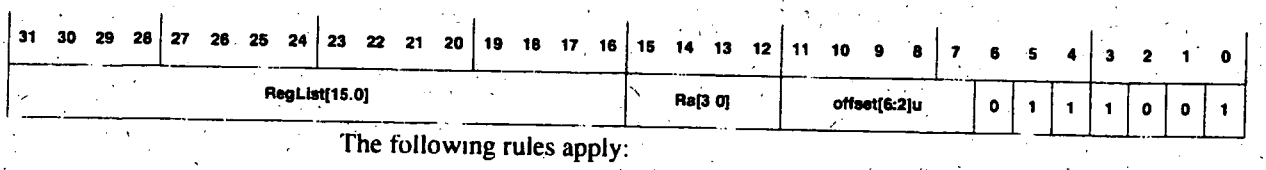
To load four registers from memory:

```
ldm {%r1, %r5, %r8, %r11}, @(16, %r12)
```

This line will generate a 16-bit instruction:

```
ldm {%r1-%r5}, @(0, %r12)
```

**32-bit Encoding**




The following rules apply:

- The meaning of RegList[15:0] is:

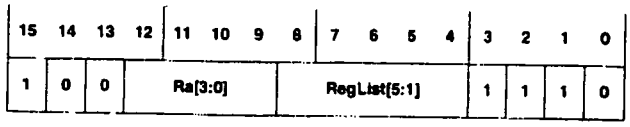
| RegList bit | register |
|---|---|
| 0 | R0 |
| 1 | R1 |

| | |
|---|---|
| 2 | R2 |
| etc | |
| 14 | R14 |
| 15 | R15 |

- The offset can take values in the range 0, 4, ..., 116, 120. An offset of 124 is reserved.

**16-bit Encoding**

The following 16-bit encoding can be used:




The following rules apply.

- The bits in RegList[5:1] have the same meaning as in the 32-bit encoding:

| RegList bit | register |
|---|---|
| 1 | R1 |
| 2 | R2 |
| 3 | R3 |
| 4 | R4 |
| 5 | R5 |

# ldm.u — Load multiple, user

**Instruction** `ldm.u RegList, @(0, Ra)`

**Description** Load multiple user registers from memory

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the final load is zero; cleared otherwise |
| **N** | Set if the result of the final load is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

`ldm.u` always loads the User Mode registers, irrespective of the processor's current operating mode.

The RegList operand specifies which registers are to be loaded and can contain any of the registers in the range R0 to R15. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 124.

The instruction is atomic.

Higher numbered registers are loaded from higher memory addresses.

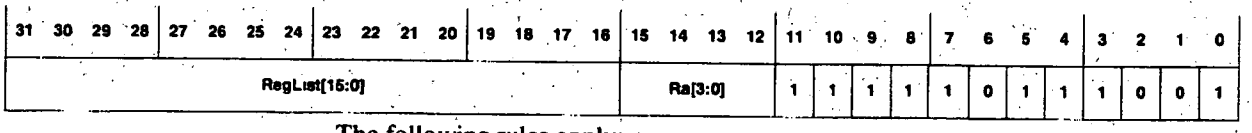
Registers are stored low register first. The flags are updated on the basis of the final load (highest register).

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples** To load four registers from memory:

```
ldm.u {%r1, %r5, %r8, %r11}, @(0, %r12)
```

**32-bit Encoding**




The following rules apply:

- The meaning of RegList[15:0] is:

| RegList bit | register |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R2 |
| etc | |
| 14 | R14 |
| 15 | R15 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ldm.x — Load multiple, exchanged order

**Instruction** `ldm.x RegList, @(offset, Ra)`

**Description** Load multiple registers from memory

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the final load is zero; cleared otherwise |
| **N** | Set if the result of the final load is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

`ldm.x` loads the registers being used by the current processor operating mode.

The RegList operand specifies which registers are to be loaded and can contain any of the registers in the range R0 to R15. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 120.

The instruction is atomic.

Higher numbered registers are loaded from lower memory addresses.

Registers are stored high register first. The flags are updated on the basis of the final load (lowest register).

A `NullPointer` exception is thrown if any of the memory addresses are zero or within the MMU's defintion of null pointers, and the instruction terminates. Loads prior to the exception complete normally.

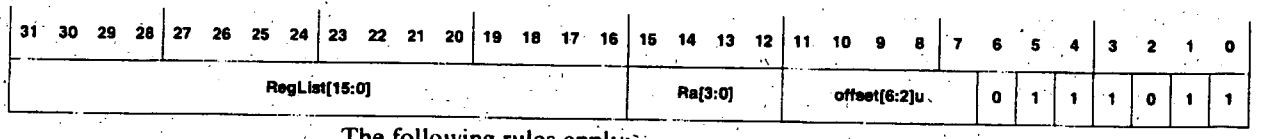
An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

To load four registers from memory:

```
ldm.x (%r1, %r5, %r8, %r11), @(16, %r12)
```

**32-bit Encoding**




The following rules apply:

- The meaning of RegList[15:0] is:

| RegList bit | Register |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R2 |
| etc | |

| | |
|---|---|
| 14 | R14 |
| 15 | R15 |

- The offset can take values in the range 0, 4, ..., 116, 120. An offset of 124 is reserved.

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# lic — Read XAP3 licence number

**Instruction** `lic Rd`

**Description** Read the XAP3's licence number

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = XAP3 licence number`

**Usage Notes** -

**Examples** `lic %r1`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3.0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mov.g — Move, GP-relative

**Instruction** `mov.g Rd, #label`

**Description** Calculate the absolute address of a GP-relative label

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = GP + #label`

**Usage Notes**

This instruction calculates the absolute address of a label and is useful in systems where a program's data segment has been loaded to an address other than zero The label would normally be a label in a data segment rather than a text segment.

The label must be aligned to a word boundary as the two least-significant bits are not encoded in the instruction.

For this instruction to produce meaningful results, a program's data segment must be loaded at the address in GP.

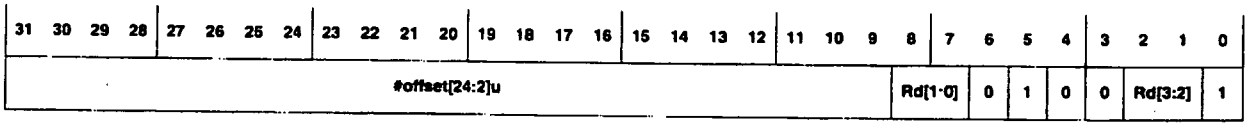
The GP register appropriate to the current processor's operating mode is used.

The label can take values between 0 and 32MB. To calculate absolute addresses for labels outside this range, the addition can be performed with add instructions.

**Examples** This instruction calculates the absolute address of `label22`:

```
mov.g %r3, #label22
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mov.h — Move, high immediate

**Instruction** `mov.h Rd, #immediate`

**Description** Move high immediate to register

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = #immediate`

**Usage Notes** This instruction can be used with `or.i` to load a 32-bit value into a register:

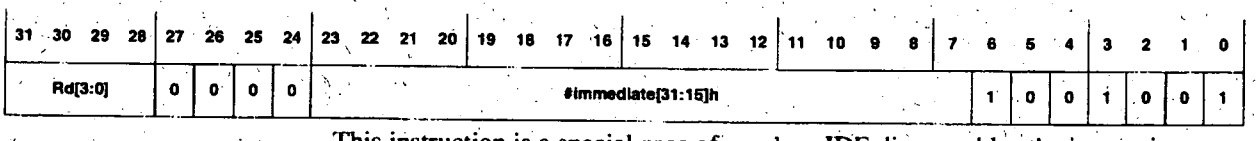
```
mov.h %r1, #0x12340000
or.i  %r1, %r1, #0x00005678
```

Refer to section "h instructions - h" encoding" for details of immediate encoding.

**Examples**

```
mov.h %r1, #0x12340000
mov.h %r1, #0x4321FFFF
```

**32-bit Encoding**




This instruction is a special case of `or.h`. xIDE disassembles the instruction into:

```
or.h Rd, %r0, #immediate
```

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mov.i — Move, immediate

**Instruction** `mov.i Rd, #immediate`

**Description** Move immediate to register

**Flags**

| | |
|---|---|
| **V** | Set if the result is zero; cleared otherwise |
| **C** | Set if the result is negative; cleared otherwise |
| **N** | Unchanged |
| **Z** | Unchanged |

**Operation** `Rd = #immediate`

**Usage Notes** This instruction can be used with `or.h` to load a 32-bit value into a register:

```
mov.i %r1, #0x00005678
or.h  %r1, %r1, #0x12340000
```

Refer to section "Instructions - "S" encoding" for details of immediate encoding.

**Examples**

```
mov.i %r1, #0x00001234

mov.i %r1, #0xFFFF4321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | 0 0 0 0 | #immediate[16:0]s | | | | 1 0 1 | 1 0 0 1 |

This instruction is a special case of `or.i`. xIDE disassembles the instruction into:

```
or.i Rd, %r0, #immediate
```

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd3a[2:0] 1 | 1 | #immediate[6:0]u | 0 0 1 0 |

- A #immediate[6:0]u value of zero is interpreted as -1. Hence, immediate values in the range -1, 1, 2, ..., 127 can be expressed.
- The meaning of the 3-bit Rd3a field is:

| Rd3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |

| 111 | R7 |
|---|---|

# mov.p — Move, PC-relative

**Instruction** `mov.p Rd, #label`

**Description** Calculate the absolute address of a label

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = #label`

**Usage Notes**

This instruction calculates the absolute address of a label and is useful in systems where a program's text segment has been loaded to an address other than zero. The label would normally be a label in a text segment rather than a data segment.

The label must be aligned to a half-word boundary as the least-significant bit is not encoded in the instruction.

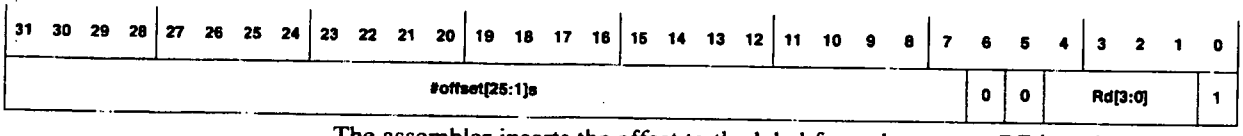
The label can be between -32MByte to +32MByte relative to the current PC. To calculate absolute program addresses outside this range, it is necessary to know the actual text segment start address.

**Examples**

This instruction calculates the absolute address of `label36`:

```
mov.p %r3, #label36
```

**32-bit Encoding**




The assembler inserts the offset to the label from the current PC into the instruction.

The following rules apply:

- Rd cannot be R15 or R0

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mov.r — move, register

**Instruction** `mov.r Rd, Rt`

**Description** Register-to-register move

**Flags**

| | |
|---|---|
| V | Set if the result is zero; cleared otherwise |
| C | Set if the result is negative, cleared otherwise |
| N | Unchanged |
| Z | Unchanged |

**Operation** `Rd = Rt`

**Usage Notes** -

**Examples** `mov.r %r1, %r2`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | 0 0 0 0 | 1 1 1 1 | 1 1 1 1 | 0 1 0 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

This instruction is a special case of `or.r`. xIDE disassembles the instruction into:

```
or.r Rd, %r0, Rt
```

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | Rt[3:0] | 1 0 1 1 | 0 0 0 0 |

The following rules apply:

- Rd cannot be R0

# movb2r — Move breakpoint to register

**Instruction** `movb2r Rd, BRs`

**Description** Transfer breakpoint register Bs to register Rd

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = BRs`

**Usage Notes** This instruction throws an `IllegalInstruction` exception if executed in User Mode.

To access the BRKE register, use the `movr2s` instruction.

**Examples** `movb2r %r1, %brk2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | BRs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

The following rules apply:

- The meaning of BRs[3:0] is:

| BRs | register |
|---|---|
| 0 | `%brk0` |
| etc | |
| 7 | `%brk7` |
| 8 | `%brk4count` |
| 9 | `%brk5count` |
| 10 | `%brk6count` |
| 11 | `%brk7count` |
| 12 | `%brk6mask` |
| 13 | `%brk7mask` |
| 14 | `%brk6data` |
| 15 | `%brk7data` |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# movm — Move multiple

**Instruction**

```
movm Rs1/#immediate1,
     Rs2/#immediate2,
     Rs3/#immediate3,
     Rs4/#immediate4,
     Rs5/#immediate5
```

**Description** Move multiple values or registers into registers R1 to R5, R1 first

**Flags**

| | |
|---|---|
| Z | Set if the result of the last move is zero; cleared otherwise |
| N | Set if the result of the last move is negative; cleared otherwise |
| C | Unchanged |
| V | Unchanged |

**Operation** -

**Usage Notes** This instruction allows each of registers R1-R5 to be:

- Loaded with an immediate value in the range -1 to +15, or
- Loaded from another register, or
- Unchanged
- The five operands following the mnemonic control the value written to R1 to R5.
- Allowed immediate values are -1, 1, 2, ... 15. To load a register with zero, specify R0 as the source register.

**Examples** To copy R7 to R1, 7 to R2, R10 to R3, R3 to R4, R12 to R5 (in that order):

```
movm %r7, #7, %r10, %r3, %r12
```

To copy R9 to R2, then –1 to R3:

```
movm %r1, %r9, #-1, %r4, %r5
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs5[3:0] / #imm5[3:0] | Rs4[3:0] / #imm4[3:0] | Rs3[3:0] / #imm3[3:0] | Rs2[3:0] / #imm2[3:0] | Rs1[3:0] / #imm1[3:0] | flags[5:1] | | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

The bits in flags control how each of the Rs fields is interpreted.

- A flag bit of 1 indicates that the corresponding Rs field is to be interpreted as an immediate value
- A flag bit of 0 indicates that the corresponding Rs field is to be treated as a source register specification

Immediate values are encoded as follows:

- An immediate value of -1 is represented by 0
- Immediate values of 1 to 15 are represented by 1 to 15

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | rdpr[2 0] | | | RsB[3:0] | | | | RsA[3.0] | | | | 1 | 0 | 1 | 0 |

The 16-bit version of the instruction can copy to a limited selection of registers. It copies.

- RsA to RdA
- RsB to RdB

RsA and RsB are defined as follows:

| RsA, RsB | meaning |
|---|---|
| 0 | R0 |
| etc | |
| 7 | R7 |
| 8 | R8 |
| 9 | R9 |
| 10 | R10 |
| 11 | R15 |
| 12 | #4 |
| 13 | #1 |
| 14 | #2 |
| 15 | #3 |

RdA and RdB are defined as follows:

| rdpr | RdB | RdA |
|---|---|---|
| 0 | R2 | R1 |
| 1 | R3 | R1 |
| 2 | R4 | R1 |
| 3 | R5 | R1 |
| 4 | R3 | R2 |
| 5 | R4 | R2 |
| 6 | R4 | R3 |
| 7 | R5 | R4 |

# movm.x — Move multiple, exchanged order

**Instruction**

```
movm.x Rs1/#immediate1,
       Rs2/#immediate2,
       Rs3/#immediate3,
       Rs4/#immediate4,
       Rs5/#immediate5
```

**Description** Move multiple values or registers into registers R1 to R5, R5 first

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the last move is zero; cleared otherwise |
| **N** | Set if the result of the last move is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes** This instruction allows each of registers R1-R5 to be:

- Loaded with an immediate value in the range -1 to +15, or
- Loaded from another register
- Unchanged
- The five operands following the mnemonic control the value written to R1 to R5.
- Allowed immediate values are -1, 1, 2, . 15. To load a register with zero, specify R0 as the source register.
- The instruction is identical to movm except that R5 is loaded first.

**Examples** To copy R12 to R5, R3 to R4, R10 to R3, 7 to R2, R7 to R1 (in that order):

```
movm.x  %r7, #7, %r10, %r3, %r12
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs5[3:0] / #imm4[3:0] | Rs4[3:0] / #imm4[3:0] | Rs3[3:0] / #imm3[3:0] | Rs2[3:0] / #imm2[3:0] | Rs1[3:0] / #imm1[3:0] | flags[5:1] | | 0 | 1 | 0 | 1 | 0 | 1 | 1 |

The bits in flags control how each of the Rs fields is interpreted.

- A flag bit of 1 indicates that the corresponding Rs field is to be interpreted as an immediate value
- A flag bit of 0 indicates that the corresponding Rs field is to be treated as a source register specification

Immediate values are encoded as follows:

- An immediate value of -1 is represented by 0
- Immediate values of 1 to 15 are represented by 1 to 15

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# movr2b — Move register to breakpoint

**Instruction** `movr2b BRd, Rs`

**Description** Transfer a register to a breakpoint register

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `BRd = Rs`

**Usage Notes** This instruction throws an `IllegalInstruction` exception if executed in User Mode.

To access the BRKE register, use the `movr2s` instruction.

**Examples** `movr2b %brk2, %r1`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| BRd[3 0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 1 1 0 | 0 0 1 0 | 0 1 1 1 | 1 1 1 1 |

The following rules apply:

- The meaning of BRd[3:0] is:

| BRd | register |
|---|---|
| 0 | %brk0 |
| etc | |
| 7 | %brk7 |
| 8 | %brk4count |
| 9 | %brk5count |
| 10 | %brk6count |
| 11 | %brk7count |
| 12 | %brk6mask |
| 13 | %brk7mask |
| 14 | %brk6data |
| 15 | %brk7data |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# movr2s — Move register to special

**Instruction** `movr2s SRd, Rs`

**Description** Write to a special register

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `SRd = Rs`

**Usage Notes**

The `movs2r` and `movr2s` instructions allow Interrupt, Supervisor and NMI Mode access to the Special Registers.

The `movr2s` instruction can be used to change from any privileged mode to any mode.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples** `movr2s %flags_i, %r1`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| SRd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 1 1 0 | 0 0 0 0 | 0 1 1 1 | 1 1 1 1 |

The following rules apply:

- The meaning of SRd[3:0] is:

| SRs | register |
|---|---|
| 0 | %flags |
| 1 | %sflags_s |
| 2 | %sflags_i |
| 3 | %sflags_n |
| 4 | %gp_u |
| 5 | %gp_s |
| 6 | %gp_i |
| 7 | %ivtb |
| 8 | %brke |
| 9-15 | reserved |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# movs2r — Move special to register

**Instruction** `movs2r Rd, SRs`

**Description** Read from a special register

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero, cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

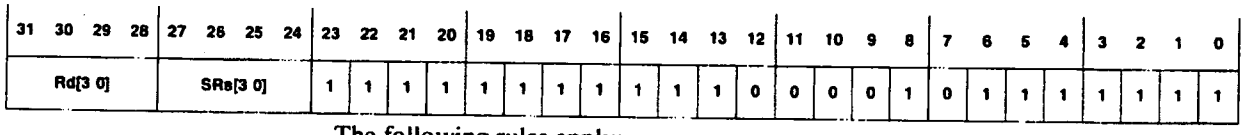
**Operation** `Rd = SRs`

**Usage Notes** The `movs2r` and `movr2s` instructions allow Interrupt, Supervisor and NMI Mode access to the Special Registers.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples** `movs2r %r1, %flags_i`

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | SRs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 1 1 0 | 0 0 0 1 | 0 1 1 1 | 1 1 1 1 |

The following rules apply:

- The meaning of SRs[3:0] is:

| SRd | register |
|---|---|
| 0 | `%flags` |
| 1 | `%sflags_s` |
| 2 | `%sflags_i` |
| 3 | `%sflags_n` |
| 4 | `%gp_u` |
| 5 | `%gp_s` |
| 6 | `%gp_i` |
| 7 | `%ivtb` |
| 8 | `%brke` |
| 9-15 | reserved |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# msbit.r — Most significant bit register

**Instruction** `msbit.r Rd, Rs`

**Description** Find the position of the most significant bit in Rs

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

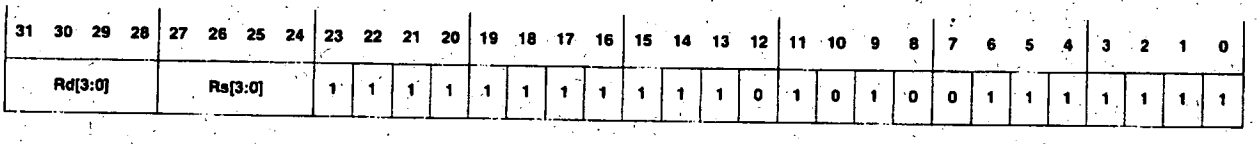
**Operation** $Rd = int(\log_2(Rs))$

**Usage Notes** Rd is set with the position of the highest 1 in the word. It should be used on positive or unsigned numbers. The value returned in Rd is (1 + highest bit to contain a 1 in Rs). This is the minimum number of bits that must be shifted to the right to make Rs = 0

**Examples**

```
msbit.r %r1, %r2
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 1 1 0 | 1 0 1 0 | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction

# mult.16s.i — Multiply, 16-bit signed, immediate

**Instruction** `mult.16s.i Rd, Rs, #immediate`

**Description** 16-bit by 16-bit signed multiply to give 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs[15:0] * #immediate`

**Usage Notes** Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767. Bits [31:16] of Rs and the immediate are ignored.

The overflow (V) bit is set only when both operands are -32768 (0x8000).

**Examples** `mult.16s.i %r1, %r2, #0x00001234`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | #immediate[15:0] | | | 1 0 0 | 1 1 0 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mult.16s.r — Multiply, 16-bit signed, register

**Instruction** `mult16.s.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit signed multiply to give 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs[15:0] * Rt[15:0]`

**Usage Notes** Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767. Bits [31:16] of Rs and Rt are ignored.

The overflow (V) bit is set only when both operands are -32768 (0x8000).

**Examples** `mult.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mult.16u.i — Multiply, 16-bit unsigned, immediate

**Instruction** `mult.16u.i Rd, Rs, #immediate`

**Description** 16-bit by 16-bit signed multiply to give 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of Rd is set; cleared otherwise |
| **C** | Cleared to zero |
| **V** | Unchanged |

**Operation** `Rd = Rs * #immediate`

**Usage Notes** Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535. Bits [31:16] of Rs and the immediate are ignored.

**Examples** `mult.16u.i %r1, %r2, #0x00001234`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | #immediate[15:0] | | | 1 0 0 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mult.16u.r — Multiply, 16-bit unsigned, register

**Instruction** `mult.16u.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit unsigned multiply to give 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of Rd is set; cleared otherwise |
| **C** | Cleared to zero |
| **V** | Unchanged |

**Operation** `Rd = Rs * Rt`

**Usage Notes** Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535. Bits [31:16] of Rs and the immediate are ignored.

**Examples** `mult.16u.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 0 1 1 1 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mult.i — Multiply immediate

| | | |
|---|---|---|
| **Instruction** | | `mult.i Rd, Rs, #immediate` |
| **Description** | | 32-bit by 32-bit multiply to give 32-bit result |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| | **V** | Set if the result of the signed operation is incorrect; cleared otherwise |
| **Operation** | | `Rd = Rs * #immediate` |
| **Usage Notes** | | This is a 32x32 multiply, giving the low 32 bits of the product in Rd. |
| | | Refer to section Instructions - "s" encoding for details of immediate encoding. |
| **Examples** | | `mult.i %r1, %r2, #0x00001234` |
| | | `mult.i %r1, %r2, #0xFFFF4321` |

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | #immediate[16:0]s | | | | 1 0 1 1 | 1 0 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# mult.r — Multiply, register

**Instruction** `mult.r Rd, Rs, Rt`

**Description** 32-bit by 32-bit multiply to give low 32-bit bits of result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero, cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs * Rt`

**Usage Notes** A 32x32 multiply gives a 64-bit result. This instruction writes the bottom 32 bits to Rd. The top 32 bits can be calculated with a `mult.h.r` instruction.

**Examples**

```
mult.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 0 0 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3.0] | Rt[3:0] | 0 1 1 1 | 0 0 0 0 |

The following rules apply:

- Rd and Rs are the same register
- Rd=0 is reserved
- Rt=0 is reserved

# mult.sh.r — Multiply, signed shift right, register

**Instruction**

```
mult.h.r Rd, Rs, Rt, #immediate
```

**Description**

32-bit by 32-bit multiply to give a 64-bit bit result which is then signed-shifted right by the immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation**

```
Rd = (Rs * Rt) >> #immediate
```

**Usage Notes**

The 32x32 multiply of Rs and Rt produces a 64-bit product in a temporary register. The 64-bit product is signed-shifted right by the immediate operand. This shift, like `shiftr.64s.i`, preserves the sign of the product by re-inserting the sign bit into the vacated most-significant bit.

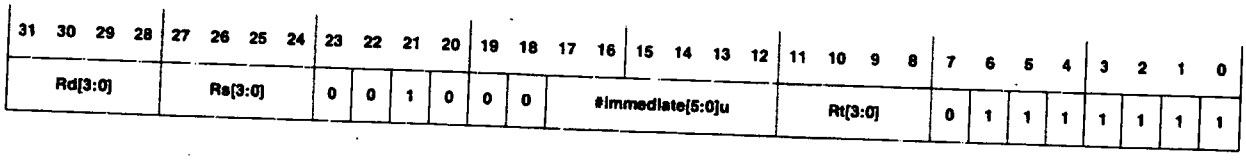
Finally, the low 32 bits are written to Rd.

This instruction is useful when operating on fixed-point numbers, where a normalising shift is needed after a multiplication.

**Examples**

```
mult.sh.r %r1, %r2, %r3, #12
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | 0 | 1 | 0 | 0 | 0 | #immediate[5:0]u | | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

## nop — No operation

| | | |
|---|---|---|
| **Instruction** | | nop |
| **Description** | | No operation |
| **Flags** | **Z** | Unchanged |
| | **N** | Unchanged |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | - |
| **Usage Notes** | | - |
| **Examples** | | nop |

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# or.h — Or, high immediate

| | | |
|---|---|---|
| **Instruction** | | `or.h Rd, Rs, #immediate` |
| **Description** | | Logical OR of Rs with a high immediate value |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs \| #immediate` |
| **Usage Notes** | | Refer to section "h instructions - "h" encoding" for details of immediate encoding. |
| **Examples** | | `or.h %r1, %r2, #0x12340000` |
| | | `or.h %r1, %r2, #0x4321FFFF` |

**32-bit Encoding**

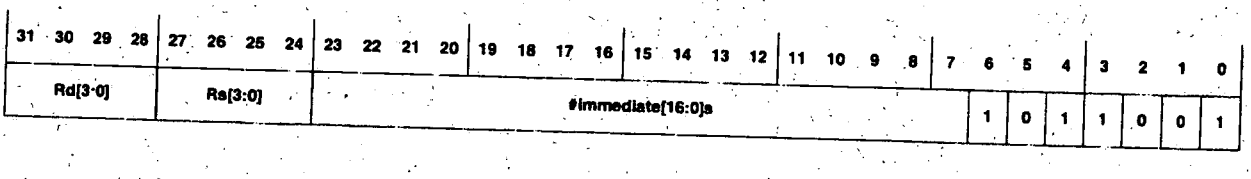
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | #immediate[31:15]h | | | | 1 0 0 | 1 0 0 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## or.i — Or, immediate

| | | |
|---|---|---|
| **Instruction** | | `or.i Rd, Rs, #immediate` |
| **Description** | | Logical OR of Rs with an immediate value |
| **Flags** | **Z** | Set if the result is zero, cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs \| #immediate` |
| **Usage Notes** | | Refer to section "Instructions "S" encoding" for details of immediate encoding. |
| **Examples** | | `or.i %r1, %r2, #0x00001234` |
| | | `or.i %r1, %r2, #0xFFFF4321` |

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# or.r

## Or, register

**Instruction** `or.r Rd, Rs, Rt`

**Description** Logical OR of two registers

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = Rs | Rt`

**Usage Notes** The XAP3 does not explicitly support a register-to-register move instruction. However, this can be performed with `or.r` and R0:

```
or.r  %r9, %r0, %r11     // Copy R11 to R9
```

**Examples**

```
or.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3.0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rt[3:0] | | | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

The following rules apply:

- Rd and Rs are the same register
- Rd=0 is reserved
- Rt=0 is reserved

# pop

## Pop from stack

**Instruction**

```
pop #immediate, RegList, #offset
pop Rs,         RegList, #offset
```

**Description**

Pop registers from stack and load a return value in R1

**Flags**

| | |
|---|---|
| **Z** | Set if the return value is zero; cleared otherwise |
| **N** | Set if the return value is negative; cleared otherwise |
| **C** | Cleared to zero |
| **V** | Cleared to zero |

**Operation**

-

**Usage Notes**

This instruction can be used to close a stack frame and specify a return value from a subroutine. The return value can be an immediate in the range -1 to +15, or a value from a register. In addition to popping registers off the stack, the stack pointer can be increased by a further amount to remove the callee's local variables.

The RegList operand specifies which registers are popped and can contain any of the registers in the range R6 to R14. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 4092.

The operation sequence is:

- The return value (from a register or an immediate value) is copied to R1
- Registers R6 to R14 are loaded from the stack, as specified in RegList. R6 is loaded from the lowest memory address
- The stack pointer is increased by `#offset+4n`, where n is the number of selected registers in RegList.
- The flags are set based on a comparison of R1 with zero.

Registers R1 to R5 cannot be loaded with pop. These registers are not normally preserved over a function call.

To return a value of zero from a function, specify R0 as the return value.

`RegList` does not convey the order in which the registers are popped from the stack.

Refer to section "Instructions - "u" encoding" for details of immediate encoding.

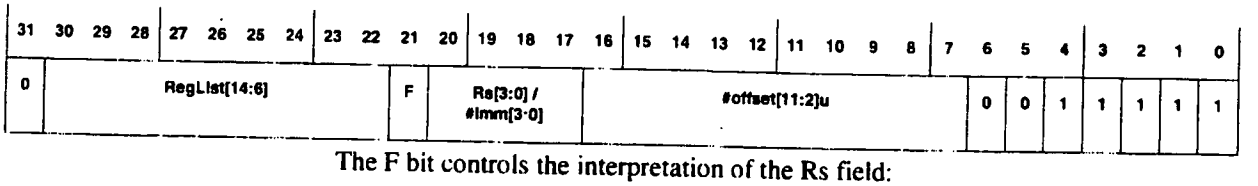
A `NullPointer` exception is thrown if the stack pointer ascends to zero or to within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned.

**Examples**

```
pop   #7, {%r6, %r7, %r8, %r14}, #0

pop   %r9, {%r6, %r7, %r8, %r14}, #8
```

### 32-bit Encoding



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | RegList[14:6] | | | | | | | | | F | Rs[3:0] / #Imm[3:0] | | | | | #offset[11:2]u | | | | | | | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | |

The F bit controls the interpretation of the Rs field:

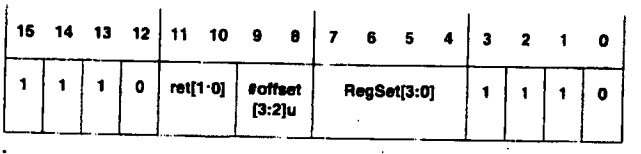
- An F bit of 1 indicates that the Rs field contains an immediate value
- An F bit of 0 indicates that the Rs field contains a register specification

Immediate values in the Rs field are encoded as follows:

- An immediate value of -1 is represented by 0
- Immediate values of 1 to 15 are represented by 1 to 15

### 16-bit Encoding

The following 16-bit encoding can be used:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | ret[1:0] | | #offset [3:2]u | | RegSet[3:0] | | | | 1 | 1 | 1 | 0 |

The following rules apply:

- The ret field specifies the return value and is:

| ret[1:0] | return value |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R6 |
| 3 | #1 |

- The RegSet field specifies one of the following register sets:

| RegSet | registers |
|---|---|
| 0 | {} |
| 1 | {R14} |
| 2 | {R6} |
| 3 | {R6, R14} |
| 4 | {R6, R7} |
| 5 | {R6, R7, R14} |
| 6 | {R6, R7, R8} |
| 7 | {R6, R7, R8, R14} |
| 8 | {R6, R7, R8, R9} |
| 9 | {R6, R7, R8, R9, R14} |
| 10 | {R6, R7, R8, R9, R10} |
| 11 | {R6, R7, R8, R9, R10, R14} |
| 12 | {R6, R7, R8, R9, R10, R11} |
| 13 | {R6, R7, R8, R9, R10, R11, R14} |

| | |
|---|---|
| 14 | {R6, R7, R8, R9, R10, R11, R12, R14} |
| 15 | {R6, R7, R8, R9, R10, R11, R12, R13, R14} |

# pop.ret — Pop from stack and return

**Instruction**

```
pop.ret #immediate, RegList, #offset
pop.ret Rs,         RegList, #offset
```

**Description** Pop registers from stack and return with a return value in R1

**Flags**

| | |
|---|---|
| **Z** | Set if the return value is zero; cleared otherwise |
| **N** | Set if the return value is negative, cleared otherwise |
| **C** | Cleared to zero |
| **V** | Cleared to zero |

**Operation** -

**Usage Notes**

This instruction is identical to pop but additionally returns from a subroutine

The return value can be an immediate in the range -1 to +15, or a value from a register. In addition to popping registers off the stack, the stack pointer can be increased by a further amount to remove the callee's local variables.

The RegList operand specifies which registers are popped and can contain any of the registers in the range R6 to R14. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 4092.

The operation sequence is:

- The return value (from a register or an immediate value) is copied to R1
- Registers R6 to R14 are loaded from the stack, as specified in RegList. R6 is loaded from the lowest memory address
- The stack pointer is increased by `#offset+4n`, where n is the number of selected registers in RegList.
- The flags are set based on a comparison of R1 with zero.
- The Link Register (R14) is copied to the Program Counter. This is done even if R14 is not in RegList.

Registers R1 to R5 cannot be loaded with pop.ret. These registers are not normally preserved over a function call.

To return a value of zero from a function, specify R0 as the return value

`RegList` does not convey the order in which the registers are popped from the stack.

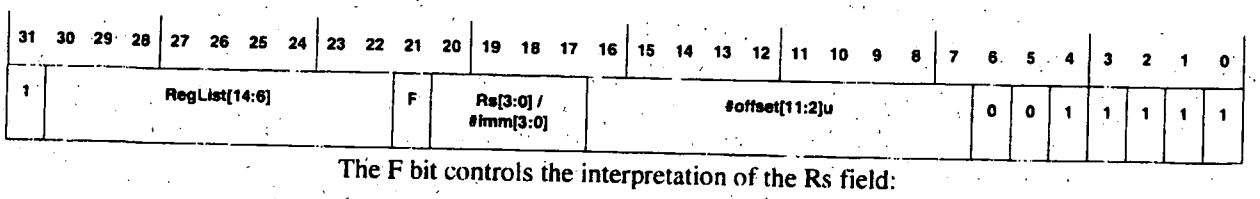
Refer to section "Instructions - "u" encoding" for details of immediate encoding.

A `NullPointer` exception is thrown if the stack pointer ascends to zero or to within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU. An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned. An `IllegalCodeAddress` exception is thrown if bit 0 of R14 is not zero.

**Examples**

```
pop.ret #-1, {%r6, %r7, %r8, %r14}, #0
pop.ret %r11, {%r6, %r7, %r8, %r14}, #8
```

**32-bit Encoding**




The F bit controls the interpretation of the Rs field:

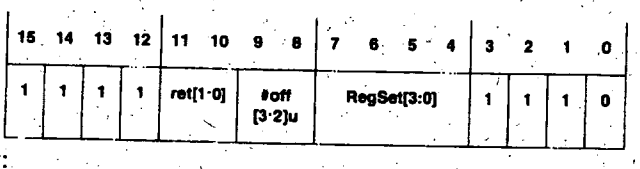
- An F bit of 1 indicates that the Rs field contains an immediate value
- An F bit of 0 indicates that the Rs field contains a register specification

Immediate values in the Rs field are encoded as follows:

- An immediate value of -1 is represented by 0
- Immediate values of 1 to 15 are represented by 1 to 15

**16-bit Encoding**

The following 16-bit encoding can be used:




The following rules apply:

- The ret field specifies the return value and is:

| ret[1:0] | return value |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R6 |
| 3 | #1 |

- The RegSet field specifies one of the following register sets:

| RegSet[3:0] | registers |
|---|---|
| 0 | {} |
| 1 | {R14} |
| 2 | {R6} |
| 3 | {R6, R14} |
| 4 | {R6, R7} |
| 5 | {R6, R7, R14} |
| 6 | {R6, R7, R8} |
| 7 | {R6, R7, R8, R14} |
| 8 | {R6, R7, R8, R9} |
| 9 | {R6, R7, R8, R9, R14} |
| 10 | {R6, R7, R8, R9, R10} |
| 11 | {R6, R7, R8, R9, R10, R14} |

| | |
|---|---|
| 12 | {R6, R7, R8, R9, R10, R11} |
| 13 | {R6, R7, R8, R9, R10, R11, R14} |
| 14 | {R6, R7, R8, R9, R10, R11, R12, R14} |
| 15 | {R6, R7, R8, R9, R10, R11, R12, R13, R14} |

# print.r — Print, register

**Instruction** `print.r Rs`

**Description** Print a register (used in conjunction with a debugger)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes** The xIDE simulator prints value the in Rs[7:0], interpreted as a character. This can be used for generating putchar() output in the debugger.

Hardware implementations of XAP3 treat this instruction as a nop.

**Examples** `print.r %r1`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# push — Push to stack

**Instruction** `push RegList, #offset`

**Description** Push registers onto the stack

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

The offset can take values in the range 0, 4, ..., 4088, 4092.

The RegList operand specifies which registers are to be loaded and can contain any of the registers in the range R1 to R14. Refer to section Register Lists for details on RegList specifications.

The operation sequence is:

- Decrease the Stack Pointer (R15) by `#offset+4n`, where n is the number of selected registers.
- Store the selected registers to memory. R14 is stored first, to the highest address. R1 is stored last, to the lowest address.

Registers are pushed onto the stack highest registers first.

Refer to section "Instructions - "u" encoding" for details of immediate encoding.

A `NullPointer` exception is thrown if the stack pointer descends to zero or within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU. An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned.

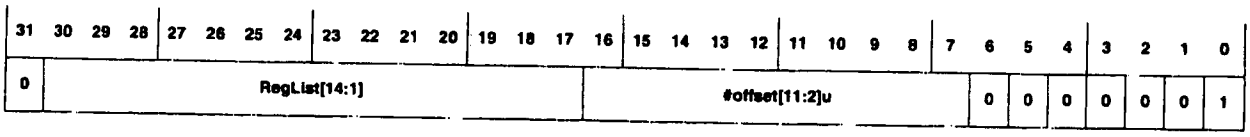
**Examples**

To save registers R1 to R14 and allocate one word on the stack:

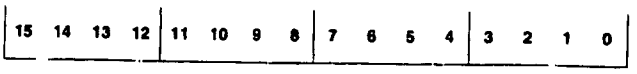
```
push {%r1-%r14}, #4

push {%r6, %r7, %r8, %r14}, #0
```

**32-bit Encoding**




**16-bit Encoding** For push operations of the form `push {%r0-%r5}, #0`, the following 16-bit encoding can be used:



| 1 | 1 | 0 | RegMask[5:1] | | | | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

The following rules apply:

- Only registers R1 to R5 can be pushed.
- The offset must be zero.

# push.r — Push to stack and registers

**Instruction** `push.r RegList, #offset`

**Description** Push registers to other registers and onto the stack

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the final push is zero; cleared otherwise |
| **N** | Set if the result of the result of the final push is negative; cleared |
| **C** | Unchanged otherwise |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

The offset can take values in the range 0 to 4092.

The RegList operand specifies which registers are to be loaded and can contain any of the registers in the range R1 to R14. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The operation sequence is:

- Decrease the Stack Pointer (R15) by `#offset+4n`, where n is the number of selected registers.
- Store the selected R6-R14 registers to memory. R14 is stored first, to the highest address. R6 is stored last, to the lowest address.
- Copy selected R1-R5 registers to selected R6-R14, starting with R1.
- Set the Z and N flags according to the result of the last copy operation. If there is no copy (none of R1-R5 are selected in RegList), the flags are unchanged.

Registers are pushed onto the stack highest registers first.

If there are more registers in the range R1-R5 than there are in the range R6-R14, the processor copies as many R1-R5 registers as it can to the R6-R14 range, starting with lower-numbered registers. Registers from the R1-R5 range not written to R6-R14 are not moved. The XAP3 assembler does not permit this condition to occur.

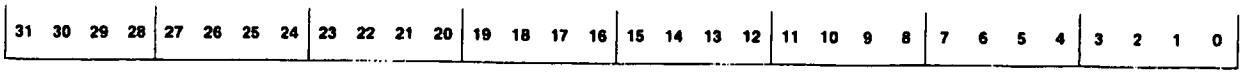
Refer to section "Instructions - 'u' encoding" for details of immediate encoding.

A `NullPointer` exception is thrown if the stack pointer descends to zero or within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned.

**Examples**

```
push.r {%r1, %r3, %r6, %r7, %r8, %r14}, #0
```

**32-bit Encoding**



| 1 | RegList[14:1] | #offset[11:2]u | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|

**16-bit Encoding**

The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | RegMask[5:1] | | | | | RegSet[3:0] | | | | 1 | 1 | 1 | 0 |

The RegMask field has one bit for each of R1 to R5, with the lowest bit selecting R1.

The offset must be zero.

The RegSet field specifies one of the following register sets:

| RegSet[3:0] | registers |
|---|---|
| 0 | {} |
| 1 | {R14} |
| 2 | {R6} |
| 3 | {R6, R14} |
| 4 | {R6, R7} |
| 5 | {R6, R7, R14} |
| 6 | {R6, R7, R8} |
| 7 | {R6, R7, R8, R14} |
| 8 | {R6, R7, R8, R9} |
| 9 | {R6, R7, R8, R9, R14} |
| 10 | {R6, R7, R8, R9, R10} |
| 11 | {R6, R7, R8, R9, R10, R14} |
| 12 | {R6, R7, R8, R9, R10, R11} |
| 13 | {R6, R7, R8, R9, R10, R11, R14} |
| 14 | {R6, R7, R8, R9, R10, R11, R12, R14} |
| 15 | {R6, R7, R8, R9, R10, R11, R12, R13, R14} |

# rem.s.r — Remainder, signed, register

**Instruction** `rem.s.r Rd, Rs, Rt`

**Description** 32-bit by 32-bit signed divide to give 32-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = Rs % Rt`

**Usage Notes** Attempts to divide by zero (Rt = 0) operate as follows:

- In User Mode, the processor throws a `DivideByZero` exception.
- In one of the privileged modes, the division does not take place and the destination register remains unchanged.

The remainder 0x80000000 % -1 gives a result of 0 and sets the overflow flag.

**Examples** `rem.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 1 1 0 0 | Rt[3 0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# rem.u.r — Remainder, unsigned, register

**Instruction** `rem.u.r Rd, Rs, Rt`

**Description** 32-bit by 32-bit unsigned divide to give 32-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation** `Rd = Rs % Rt`

**Usage Notes** Attempts to divide by zero (Rt=0) operate as follows:

- In User Mode, the processor throws a `DivideByZero` exception.
- In one of the privileged modes, the division does not take place and the destination register remains unchanged.

**Examples**

```
rem.u.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# rlinm.s Rotate left, immediate and mask, sign-extend

**Instruction** `rlinm.s Rd, Rs, #shift, #highbit, #lowbit`

**Description** Rotate left immediate, then and with mask and sign-extend

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the operation is zero; cleared otherwise |
| **N** | Set if the result of the operation is negative; cleared otherwise |
| **C** | The C bit is set to bit 0 of Rs after the rotation but before the masking operation |
| **V** | Unchanged |

**Operation** -

**Usage Notes** shift, highbit and lowbit can take values in the range 0 to 31.

`Rs` is first rotated to the left by `#shift`.

The result is then ANDed with a 32-bit mask that consists of:

- Contiguous string of ones between (and including) `#highbit` and `#lowbit`
- Zeroes for all other bits

The result is then sign-extended from #highbit to bit 31 before being stored in Rd.

`#highbit` must be greater than or equal to `#lowbit`.

This instruction can be used to implement other commonly found instructions:

- A signed shift by an immediate value
- Sign extension

**Examples** `rlinm.s %r4, %r6, #0, #15, #31`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | 1 | #shift[4:0] | | #highbit[4 0] | #lowbit[4.0] | 1 0 1 1 | 1 1 1 1 |

**16-bit Encoding – 1** A signed shift by immediate "`shiftr.s.i Rd, Rd, #(32-imm)&31`" operation can be implemented with `rlinm.s`:

| 15 | 14 | 13 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | Rd3e[2:0] | #imm[4:0] | 1 | 1 | 1 | 0 |

This instruction implements:

`rlinm.s #imm, #(imm - 1), #0`

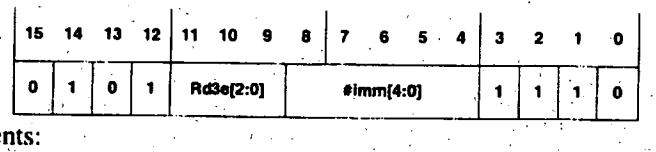
- The meaning of the 3-bit Rd3e field is:

| Rd3e | register |
|---|---|

| | |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

**16-bit Encoding – 2**

A sign-extending "`sext.i Rd, #imm`" operation can be implemented with rlinm.s:




This instruction implements:

```
rlinm.s #0, #imm, #0
```

- The meaning of the 3-bit Rd3e field is:

| Rd3e | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

# rlinm.z Rotate left, immediate and mask, zero-extend

**Instruction** `rlinm.z Rd, Rs, #shift, #highbit, #lowbit`

**Description** Rotate left immediate, then and with mask and sign-extend

**Flags**

| | |
|---|---|
| **Z** | Set if the result of the operation is zero; cleared otherwise |
| **N** | Set if the result of the operation is negative; cleared otherwise |
| **C** | The C bit is set to bit 0 of Rs after the rotation but before the masking operation |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

shift, highbit and lowbit can take values in the range 0 to 31.

`Rs` is first rotated to the left by `#shift`.

The result is then ANDed with a 32-bit mask that consists of:

- Contiguous string of ones between (and including) `#highbit` and `#lowbit`
- Zeroes for all other bits

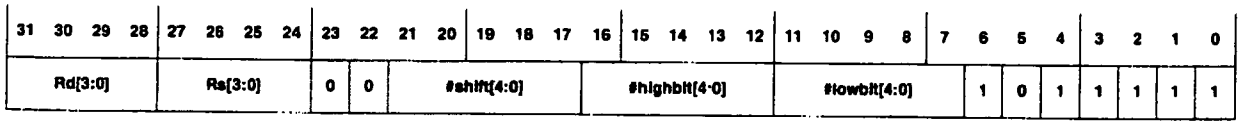
`#highbit` must be greater than or equal to `#lowbit`.

This instruction can be used to implement other commonly found instructions:

- An unsigned shift by an immediate value
- An AND with an immediate value

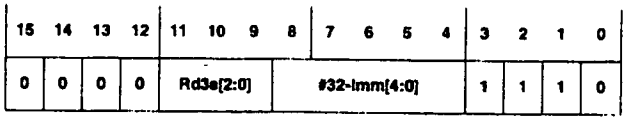
**Examples** `rlinm.z %r4, %r6, #0, #15, #31`

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 | 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 0 | #shift[4:0] | | #highbit[4:0] | #lowbit[4:0] | 1 0 1 | 1 1 1 1 |

**16-bit Encoding – 1** An unsigned shift by immediate "`shiftr.u.i Rd, Rd, #imm`" operation can be implemented with `rlinm.z`:



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| 0 0 0 0 | Rd3e[2:0] | #32-imm[4:0] | 1 1 1 0 |

This instruction implements:

`rlinm.z #32-imm, #31-imm, #0`

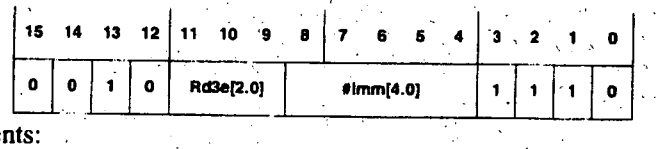
- The meaning of the 3-bit Rd3e field is:

| Rd3e | register |
|---|---|
| 000 | R8 |
| 001 | R1 |

| | |
|---|---|
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

**16-bit Encoding – 2**

An shift left by immediate "`shiftl.i Rd, Rd, #imm`" operation can be implemented with `rlinm.z`:



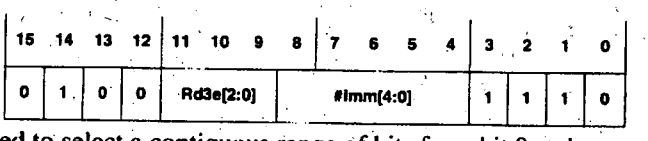

This instruction implements:

```
rlinm.z #imm, #31, #imm
```

The meaning of the 3-bit Rd3e field is given in the table above.

**16-bit Encoding – 3**

An AND with immediate "`and.i Rd, Rd, #(2^imm+1-1)`" operation can be implemented with `rlinm.z`:



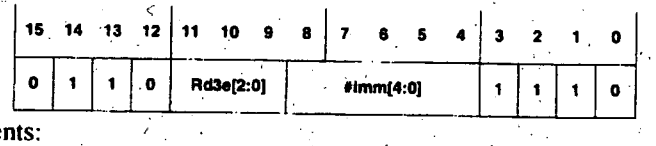

This instruction can be used to select a contiguous range of bits from bit 0 and implements:

```
rlinm.z #0, #imm, #0
```

The meaning of the 3-bit Rd3e field is given in the table above.

**16-bit Encoding – 4**

An AND with immediate "`and.i Rd, Rd, #2^imm`" operation can be implemented with `rlinm.z`:




This instruction implements:

```
rlinm.z 0, #imm, #imm
```

The meaning of the 3-bit Rd3e field is given in the table above.

# rotatel.r — Rotate left, register

**Instruction** `rotatel.r Rd, Rs, Rt`

**Description** Rotate left

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit is the final value of bit 0 of Rd |
| **V** | Unchanged |

**Operation** `Rd = (Rs << Rt[4:0]) | (Rs >> (32-Rt[4:0]))`

**Usage Notes** The bottom five bits of Rt indicate the number of places to rotate left. The rotation is a 32-bit rotation and does rotate through the carry bit.

For a rotate by an immediate value, use the `rlinm.s` or `rlinm.z` instructions.

**Examples** `rotatel %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3.0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# rtie — Return from interrupt/exception

**Instruction** `rtie`

**Description** Return from interrupt or exception

**Flags**

| | |
|---|---|
| **Z** | Restored from SFLAGS |
| **N** | Restored from SFLAGS |
| **C** | Restored from SFLAGS |
| **V** | Restored from SFLAGS |

**Operation**

```
if (FLAGS.Mode == NMIMode) then
   FLAGS = SFLAGS_N
   PC    = LR_N
elseif (FLAGS.Mode == InterruptMode) then
   FLAGS = SFLAGS_I
   PC    = LR_I
elseif (FLAGS.Mode == SupervisorMode) then
   FLAGS = SFLAGS_S
   PC    = LR_S
endif
```

**Usage Notes**

The null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code.

If nested interrupts can occur, it is necessary to save and restore SFLAGS_I within the ISR.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

This instruction throws an `IllegalCodeAddress` exception if bit 0 of the Link Register is not zero

**Examples**

```
rtie
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# scc.i — Set if carry clear, immediate

**Instruction** `scc.i Rd, Rs, #immediate`

**Description** Set register if carry clear

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.C == 0) ? Rs : #immediate`

**Usage Notes** Refer to section "i instructions – "s" encoding" for details of immediate encoding.

**Examples**

```
scc.i %r1, %r2, #0x00000123
scc.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | 1 | 0 | 1 | #immediate[12:0]s | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## scc.r — Set if carry clear, register

**Instruction** `scc.r Rd, Rs, Rt`

**Description** Set register if carry clear

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.C == 0) ? Rs : Rt`

**Usage Notes** The `scc.r` instruction is used to implement `sge.u.r Rd, Rs, Rt`.

**Examples** `scc.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3 0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# scs.i — Set if carry set, immediate

**Instruction** `scc.i Rd, Rs, #immediate`

**Description** Set register if carry set

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.C == 1) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions - "s" encoding" for details of immediate encoding.

**Examples**

```
scs.i %r1, %r2, #0x00000123
scs.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | 1 | 0 | 0 | #immediate[12:0]s | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## scs.r — Set if carry set, register

**Instruction** `scs.r Rd, Rs, Rt`

**Description** Set register if carry set

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.C == 1) ? Rs : Rt`

**Usage Notes** The `scs.r` instruction is used to implement `slt.u.r Rd, Rs, Rt`.

**Examples** `scs.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# seq.i — Set if equal, immediate

**Instruction** `seq.i Rd, Rs, #immediate`

**Description** Set register if equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

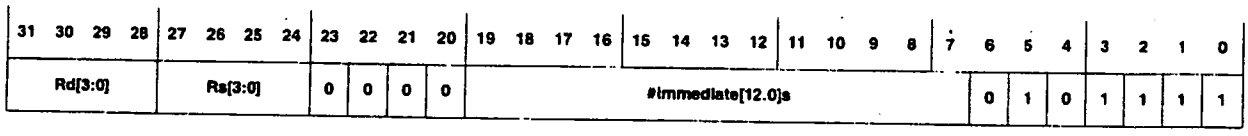
**Operation** `Rd = (FLAGS.Z == 1) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions - "S" encoding" for details of immediate encoding.

**Examples**

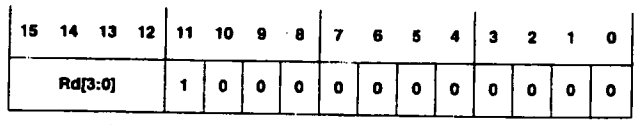
```
seq.i %r1, %r2, #0x00000123
seq.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 0 0 0 | #immediate[12.0]s | | | 0 1 0 | 1 1 1 1 |

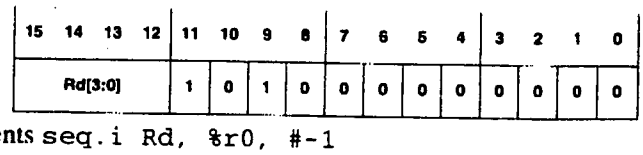
**16-bit Encoding – 1** The following 16-bit encoding can be used:



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | 1 0 0 0 | 0 0 0 0 | 0 0 0 0 |

This instruction implements `seq.i Rd, %r0, #1`

**16-bit Encoding – 2** The following 16-bit encoding can be used:



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | 1 0 1 0 | 0 0 0 0 | 0 0 0 0 |

This instruction implements `seq.i Rd, %r0, #-1`

# seq.r — Set if equal, register

**Instruction** `seq.r Rd, Rs, Rt`

**Description** Set register if equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.Z == 1) ? Rs : Rt`

**Usage Notes** -

**Examples** `seq.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 0 0 0 0 | 0 0 0 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sge.s.i Set if greater than or equal, signed, immediate

**Instruction** `sge.s.i Rd, Rs, #immediate`

**Description** Set register if greater than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N == FLAGS.V) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions – 's' encoding" for details of immediate encoding.

**Examples**

```
sge.s.i %r1, %r2, #0x00000123
sge.s.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 0 | 0 | 1 | #immediate[12:0]s | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sge.s.r Set if greater than or equal, signed, register

**Instruction** `sge.s.r Rd, Rs, Rt`

**Description** Set register if greater than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N == FLAGS.V) ? Rs : Rt`

**Usage Notes** -

**Examples** `sge.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sge.u.i Set if greater than or equal, unsigned, immediate

**Instruction** `sge.u Rd, Rs, #immediate`

**Description** Set register if greater than

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = (FLAGS.C == 0) ? Rs : #immediate`

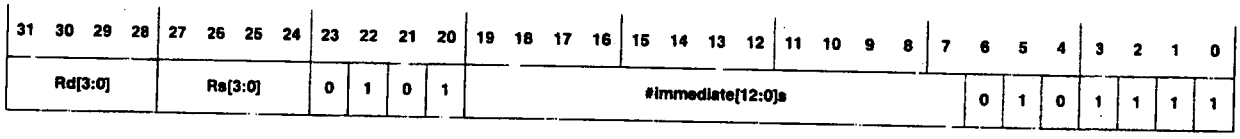
**Usage Notes** Refer to section "Instructions - "s" encoding" for details of immediate encoding.

This instruction is translated into `scc.i` by the assembler. xIDE prints this instruction as `scc.i`.

**Examples**

```
sge.u.i %r1, %r2, #0x00000234
sge.u.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sge.u.r Set if greater than or equal, unsigned, register

**Instruction** `sge.u.r Rd, Rs, Rt`

**Description** Set register if greater than or equal, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

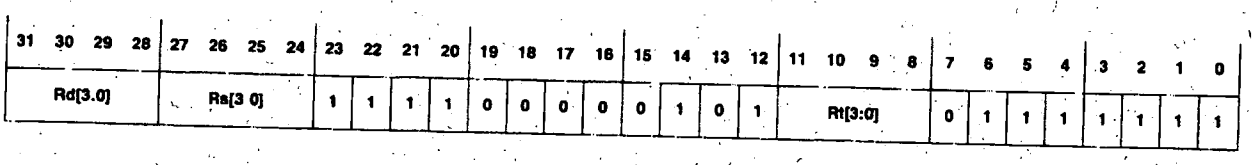
**Operation** `Rd = (FLAGS.C == 0) ? Rs : Rt`

**Usage Notes** This instruction is translated into `scc.r` by the assembler. xIDE prints this instruction as `scc.r`.

**Examples**

```
scc.r %r1, %r2, %r3
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3.0] | Rs[3 0] | 1 1 1 1 | 0 0 0 0 | 0 1 0 1 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sgt.s.i — Set if greater than, signed, immediate

**Instruction** `sgt.s.i Rd, Rs, #immediate`

**Description** Set register if greater than, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.N == FLAGS.V) && (FLAGS.Z == 0)) ? Rs :
#immediate
```

**Usage Notes** Refer to section "Instructions - "s" encoding" for details of immediate encoding.

**Examples**

```
sgt.s.i %r1, %r2, #0x00000234

sgt.s.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 0 | 1 | 1 | #immediate[12:0]s | | | 0 1 0 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## sgt.s.r — Set if greater than, signed, register

**Instruction** `sgt.s.r Rd, Rs, Rt`

**Description** Set register if greater than, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.N == FLAGS.V) && (FLAGS.Z ==0)) ? Rs : Rt
```

**Usage Notes** -

**Examples**

```
sgt.s.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3.0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sgt.u.i — Set if greater than, unsigned, immediate

**Instruction** `sgt.s.i Rd, Rs, #immediate`

**Description** Set register if greater than, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.C == 0) && (FLAGS.Z == 0)) ? Rs :
#immediate
```

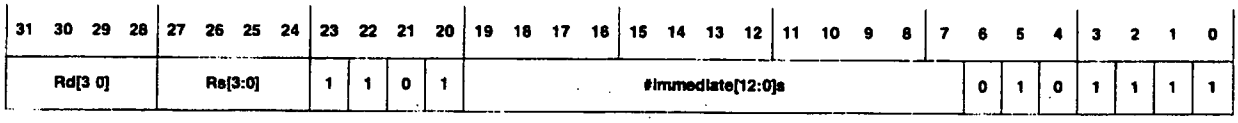
**Usage Notes** Refer to section "Instructions - "s" encoding" for details of immediate encoding.

**Examples**

```
sgt.u.i %r1, %r2, #0x00000123

sgt.u.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sgt.u.r — Set if greater than, unsigned, register

**Instruction** `sgt.u.r Rd, Rs, Rt`

**Description** Set register if greater than, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = ((FLAGS.C == 0) && (FLAGS.Z ==0)) ? Rs : Rt`

**Usage Notes** -

**Examples** `sgt.u.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rt[3.0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftl.64.i — Shift left, 64-bit, immediate

**Instruction** `shiftl Rd, #immediate`

**Description** 64-bit left shift

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} <<= #immediate`

**Usage Notes** The immediate can take values between 0 and 63.

Rd is either R8 or R10. Other registers used as Rd throw an `UnknownInstruction` exception.

**Examples** `shiftl.64.i %r8, #3`

**32-bit Encoding**

| 31 30 29 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[5:0] | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Rd[3:0] = 1000 means the 64-bit register {R9, R8}.

Rd[3:0] = 1010 means the 64-bit register {R11, R10}.

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftl.64.r — Shift left, 64-bit, register

**Instruction** `shiftl.64.r Rd, Rt`

**Description** 64-bit left shift

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} <<= Rs[5:0]`

**Usage Notes** Rd is either R8 or R10. Other registers used as Rd throw an `UnknownInstruction` exception.

Rs can be any register.

**Examples** `shiftl.64.r %r8, %r2`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | 0 1 1 1 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | Rt[3 0] | 0 1 1 1 | 1 1 1 1 |

Rd[3:0] or Rs[3:0]= 1000 means the 64-bit register {R9, R8}.

Rd[3:0] or Rs[3:0] = 1010 means the 64-bit register {R11, R10}.

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftl.r — Shift left, register

**Instruction** `shiftl.r Rd, Rs, Rt`

**Description** Shift left

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit is the final value of bit 0 of Rd |
| **V** | Unchanged |

**Operation** `Rd = Rs << Rt[4:0]`

**Usage Notes** The bottom five bits of Rt indicate the number of places to shift left.

For a shift by an immediate value, use the `rlinm.s` or `rlinm.z` instructions.

**Examples**

```
shiftl %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 0 | 1 0 1 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftr.64s.i — Shift right, 64-bit signed, immediate

**Instruction** `shiftr.64s.i Rd, #immediate`

**Description** 64-bit signed right shift

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} >>= #immediate`

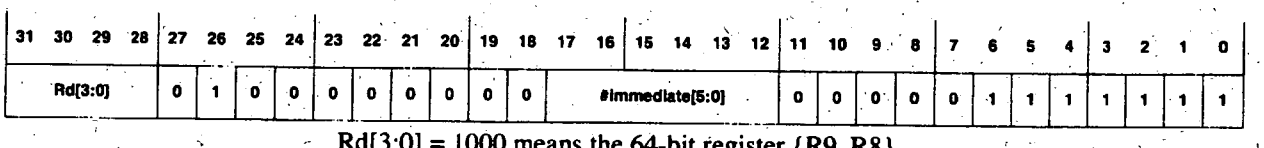
**Usage Notes** The vacated most-significant bits are filled with the sign bit.

Rd is either R8 or R10. Other registers used as Rd throw an `UnknownInstruction` exception.

The immediate can take values between 0 and 63.

**Examples** `shiftr.64s.i %r8, #63`

**32-bit Encoding**




Rd[3:0] = 1000 means the 64-bit register {R9, R8}.

Rd[3:0] = 1010 means the 64-bit register {R11, R10}.

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftr.64s.r — Shift right, 64-bit signed, register

**Instruction** `shiftr.64s.r Rd, Rt`

**Description** 64-bit signed right shift

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} >>= Rt[5:0]`

**Usage Notes** The vacated most-significant bits are filled with the sign bit.

Rd is either R8 or R10. Other registers used as Rd throw an `UnknownInstruction` exception.

**Examples** `shiftr.64.r %r8, %r2`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Rd[3:0] or Rs[3:0] = 1000 means the 64-bit register {R9, R8}.

Rd[3:0] or Rs[3:0] = 1010 means the 64-bit register {R11, R10}.

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftr.s.r — Shift right, signed, register

**Instruction** `shiftr.s.r Rd, Rs, Rt`

**Description** Signed shift right

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit is the final value of bit 0 of Rd |
| **V** | Unchanged |

**Operation** `Rd = Rs >> Rt[4:0]`

**Usage Notes** The vacated most-significant bits are filled with the sign bit.

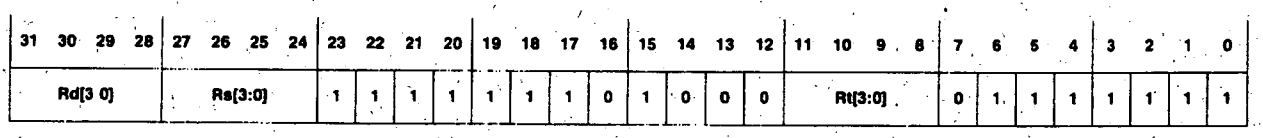
The bottom five bits of Rt indicate the number of places to shift right.

For a shift by an immediate value, use the `rlinm.s` or `rlinm.z` instructions.

**Examples**

```
shiftr.s.r %r1, %r2, %r3
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 0 | 1 0 0 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# shiftr.u.r — Shift right, unsigned, register

**Instruction** `shift.u.r Rd, Rs, Rt`

**Description** Unsigned right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit is the final value of bit 0 of Rd |
| **V** | Unchanged |

**Operation** `Rd = Rs >> Rt[4:0]`

**Usage Notes** The vacated most-significant bits are filled with zeroes.

The bottom five bits of Rt indicate the number of places to shift right.

For a shift by an immediate value, use the `rlinm.s` or `rlinm.z` instructions.

**Examples** `shiftr.u.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

**Instruction** `sif`

**Description** Perform SIF cycle

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
if (FLAGS.Mode == UserMode) then
   allow SIF access
else
   throw IllegalInstruction exception
endif
```

The `sif` instruction allows the XAP3 to perform a SIF cycle. Two SIF cycles are needed to access unaligned data objects in memory.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples** `sif`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

This encoding results in a blank flash memory appearing as `sif` instructions.

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# sle.s.i Set if less than or equal, signed, immediate

**Instruction** `sle.s.i Rd, Rs, #immediate`

**Description** Set register if less than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.N == FLAGS.V) || (FLAGS.Z == 1)) ? Rs :
#immediate
```

**Usage Notes** Refer to section "Instructions - 'S' encoding" for details of immediate encoding.

**Examples**

```
sle.s.i %r1, %r2, #0x00000123

sle.s.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 0 | 1 | 0 | #immediate[12:0]s | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sle.s.r — Set if less than or equal, signed, register

**Instruction** `sle.s.r Rd, Rs, Rt`

**Description** Set register if less than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.N == FLAGS.V) || (FLAGS.Z == 1)) ? Rs :
Rt
```

**Usage Notes** -

**Examples**

```
sle.s.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 0 0 0 0 | 1 0 1 0 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sle.u.i Set if less than or equal, unsigned, immediate

**Instruction** `sle.u.i Rd, Rs, #immediate`

**Description** Set register if less than or equal, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.C == 1) || (FLAGS.Z == 1)) ? Rs :
#immediate
```

**Usage Notes** Refer to section "Instructions – 's' encoding" for details of immediate encoding.

**Examples**

```
sle.u.i %r1, %r2, #0x00000123

sle.u.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 0 0 | #immediate[12 0]s | | | 0 1 0 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sle.u.r Set if less than or equal, unsigned, register

**Instruction** `sle.s.r Rd, Rs, Rt`

**Description** Set register if less than or equal, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = ((FLAGS.C == 1) || (FLAGS.Z == 1)) ? Rs : Rt
```

**Usage Notes** -

**Examples**

```
sle.u.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3 0] | Rs[3.0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sleepnop — Sleep

**Instruction** `sleepnop`

**Description** Put the XAP3 into NOP Sleep state

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
if (FLAGS.Mode == UserMode) then
   throw IllegalInstruction exception
else
   put XAP3 into NOP Sleep state
endif
```

**Usage Notes** Put the XAP3 into the NOP Sleep state. SIF cycles are not allowed in the NOP Sleep state. xIDE is unable to gain access to XAP3 in this state.

An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

**Examples** `sleepnop`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sleepsif — Sleep and allow SIF

**Instruction** `sleepsif`

**Description** Put the XAP3 into SIF Sleep state

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
if (FLAGS.Mode == UserMode) then
   throw IllegalInstruction exception
else
   put XAP3 into SIF Sleep mode
endif
```

**Usage Notes** Put the XAP3 into the SIF Sleep state. SIF cycles are allowed in the SIF Sleep state.

An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

**Examples** `sleepsif`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# slt.s.i — Set if less than, signed, immediate

| | | |
|---|---|---|
| **Instruction** | | `slt.s.i Rd, Rs, #immediate` |
| **Description** | | Set register if less than, signed |
| **Flags** | **Z** | Unchanged |
| | **N** | Unchanged |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = (FLAGS.N != FLAGS.V) ? Rs : #immediate` |
| **Usage Notes** | | Refer to section "Instructions - "S" encoding" for details of immediate encoding. |
| **Examples** | | `slt.s.i %r1, %r2, #0x00000123` |
| | | `slt.s.i %r1, %r2, #0xFFFFF321` |

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3 0] | Rs[3:0] | 1 | 0 | 0 | 0 | #immediate[12:0]s | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# slt.s.r — Set if less than, signed, register

**Instruction** `slt.s.r Rd, Rs, Rt`

**Description** Set register if less than, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N != FLAGS.V) ? Rs : Rt`

**Usage Notes** -

**Examples** `slt.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3 0] | Rs[3:0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# slt.u.i — Set if less than, unsigned, immediate

**Instruction** `slt.u.i Rd, Rs, #immediate`

**Description** Set register if less than

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.C == 1) ? Rs : #immediate`

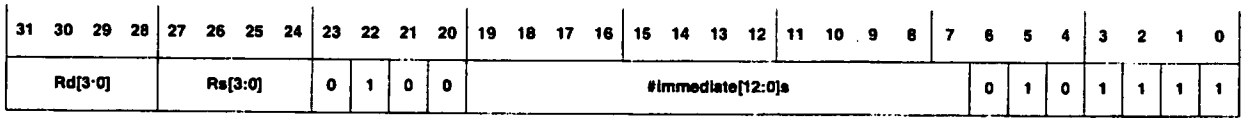
**Usage Notes** Refer to section "Instructions - "s" encoding" for details of immediate encoding.

This instruction is translated into `scs.i` by the assembler. xIDE prints this instruction as `scs.i`.

**Examples**

```
slt.u.i %r1, %r2, #0x00000123
slt.u.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 1 0 0 | #immediate[12:0]a | | | 0 1 0 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# slt.u.r — Set if less than, unsigned, register

**Instruction** `slt.u.i Rd, Rs, Rt`

**Description** Set register if less than, unsigned

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.C == 1) ? Rs : Rt`

**Usage Notes** This instruction is translated into `scs.r` by the assembler. xIDE prints this instruction as `scs.r`.

**Examples** `slt.u.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# smi.i — Set if minus, immediate

**Instruction** `smi.i Rd, Rs, #immediate`

**Description** Set register if minus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N == 1) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions - 'S' encoding" for details of immediate encoding.

**Examples**

```
smi.i %r1, %r2, #0x00000123
smi.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3.0] | Rs[3:0] | 0 | 0 | 1 | 0 | #immediate[12:0]s | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction

# smi.r — Set if minus, register

**Instruction** `smi.r Rd, Rs, Rt`

**Description** Set register if minus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N == 1) ? Rs : Rt`

**Usage Notes** -

**Examples** `smi.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sne.i — Set if not equal, immediate

**Instruction** `sne.i Rd, Rs, #immediate`

**Description** Set register if not equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.Z == 0) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions "S" encoding" for details of immediate encoding.

**Examples**

```
sne.i %r1, %r2, #0x00000123
sne.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3 0] | 0 0 0 1 | #immediate[12 0]s | | | 0 1 0 | 1 1 1 1 |

**16-bit Encoding – 1** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | 1 0 0 1 | 0 0 0 0 | 0 0 0 0 |

This instruction implements `sne.i Rd, %r0, #1`

**16-bit Encoding – 2** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | 1 0 1 1 | 0 0 0 0 | 0 0 0 0 |

This instruction implements `sne.i Rd, %r0, #-1`

## sne.r — Set if not equal, register

**Instruction** `sne.r Rd, Rs, Rt`

**Description** Set register if not equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.Z == 0) ? Rs : Rt`

**Usage Notes** -

**Examples** `sne.r %r1, %r2, #0x00001FFF`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3.0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# spl.i — Set if plus, immediate

**Instruction** `spl.i Rd, Rs, #immediate`

**Description** Set register if plus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N == 0) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions - "S" encoding" for details of immediate encoding.

**Examples**

```
spl.i %r1, %r2, #0x00000123
spl.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 0 | 0 | 1 | 1 | #immediate[12:0]s | | | | | | | | | | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# spl.r — Set if plus, register

**Instruction** `spl.r Rd, Rs, Rt`

**Description** Set register if plus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.N == 0) ? Rs : Rt`

**Usage Notes** -

**Examples** `spl.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Rt[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# st.16.i Store, 16-bit, displacement

**Instruction** `st.16.i Rs, @(offset, Ra)`

**Description** Store 16-bit value to address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

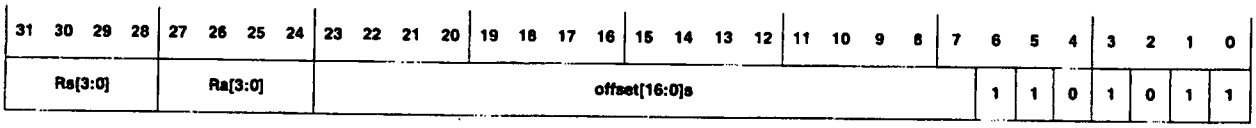
**Operation** `(* (int16*) ( offset + Ra ) ) = Rs[15:0]`

**Usage Notes** Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535.

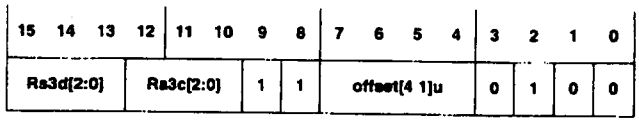
A 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU

**Examples** `st.16.i %r1, @(27, %r12)`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range 0, 2, ..., 28, 30, the following 16-bit encoding can be used:




The following rules apply:

- The meaning of the 3-bit Rs3d field is:

| Rs3d | register |
|---|---|
| 000 | R0 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

- The meaning of the 3-bit Ra3c field is:

| Ra3c | register |
|---|---|
| 000 | R15 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R8 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

# st.16.r Store, 16-bit, indexed

**Instruction** `st.16.r Rs, @(Rx, Ra)`

**Description** Store 16-bit value to address (2Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int16*) ( 2*Rx + Ra ) = Rs[15:0]`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a two-byte offset from Ra.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `st.16.r %r1, @(%r3, %r12)`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rs[3:0] | Ra[3:0] | 1 1 1 1 | 1 1 1 0 | 0 1 0 1 | Rx[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rx3a[2:0] | Ra3a[2:0] Rs2[1:0] | 1 1 0 1 | 0 0 0 0 |

The following rules apply.

- The meaning of the 3-bit Rx3a and Ra3a fields is:

| Rx3a, Ra3c | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

The meaning of the 2-bit Rs2 field is:

| Rs2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

# st.8.i — Store, 3-bit, displacement

**Instruction** `st.8.i Rs, @(offset, Ra)`

**Description** Store 16-bit value to address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

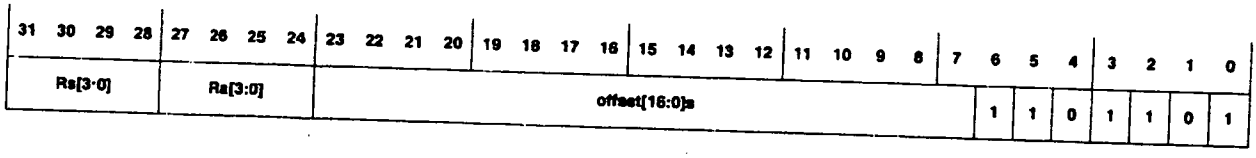
**Operation** `(* (int8*) ( offset + Ra ) ) = Rs[7:0]`

**Usage Notes** Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535.

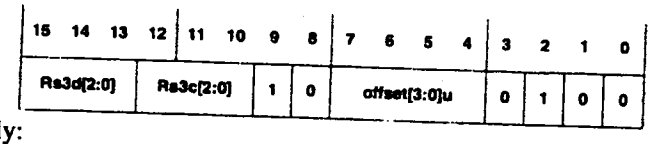
An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `st.8.i %r1, @(27, %r12)`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range 0 to 15, the following 16-bit encoding can be used:




The following rules apply:

- The meaning of the 3-bit Rs3d field is:

| Rs3d | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

The meaning of the 3-bit Ra3c field is:

| Ra3c | register |
|---|---|
| 000 | R15 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R8 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

# st.8.r — Store, 8-bit, indexed

**Instruction** `st.8.r Rs, @(Rx, Ra)`

**Description** Store 8-bit value to address (Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int8*) ( Rx + Ra ) = Rs[7:0]`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a byte offset from Ra.

An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `st.8.r %r1, @(%r3, %r12)`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ra[3:0] | | | | Ra[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rx[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rx3a[2:0] | | | | Ra3a[2:0] | | | | Rs2 [1:0] | | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

The following rules apply:

- The meaning of the 3-bit Rx3a and Ra3a fields is:

| Rx3a, Ra3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

- The meaning of the 2-bit Rs2 field is:

| Rs2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

# st.i — Store, displacement

**Instruction** `st.i Rs, @(offset, Ra)`

**Description** Store 32-bit value to address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

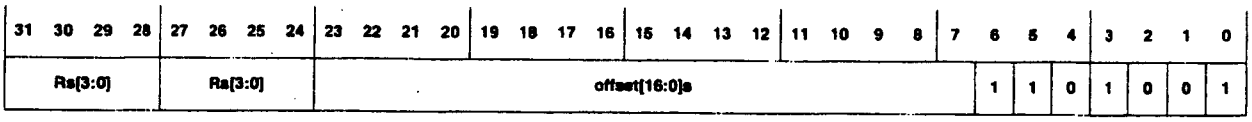
**Operation** `(* (int32*) ( offset + Ra ) ) = Rs`

**Usage Notes** Ra and offset are interpreted as byte addresses. The offset can take values in the range -65536 to +65535.

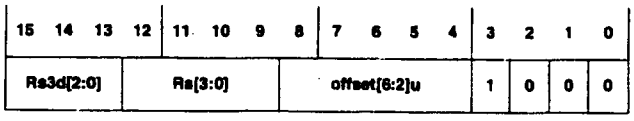
A 32-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `st.i %r1, @(27, %r12)`

**32-bit Encoding**




**16-bit Encoding** For offsets in the range 0, 4, . ., 120, 124, the following 16-bit encoding can be used:




The following rules apply:

- The meaning of the 3-bit Rs3d field is:

| Rs3d | register |
|---|---|
| 000 | R0 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R5 |
| 110 | R6 |
| 111 | R7 |

# st.r — Store, indexed

**Instruction** `st.r Rs, @(Rx, Ra)`

**Description** Store 32-bit value to address (4Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int32*) ( 4*Rx + Ra ) = Rs`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a four-byte offset from Ra.

A 32-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `st.r %r1, @(%r3, %r12)`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ra[3:0] | | | | Ra[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | Rx[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rx3a[2:0] | | | | Ra3a[2:0] | | | | Ra2[1.0] | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

The following rules apply:

- The meaning of the 3-bit Rx3a and Ra3a fields is:

| Rx3a, Ra3a | register |
|---|---|
| 000 | R8 |
| 001 | R1 |
| 010 | R2 |
| 011 | R3 |
| 100 | R4 |
| 101 | R9 |
| 110 | R6 |
| 111 | R7 |

The meaning of the 2-bit Rs2 field is:

| Rs2 | register |
|---|---|
| 00 | R4 |
| 01 | R1 |
| 10 | R2 |
| 11 | R3 |

# stm — Store multiple

**Instruction** `stm RegList, @(offset, Ra)`

**Description** Store multiple registers to memory

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

`stm` stores the registers being used by the current processor operating mode.

The RegList operand specifies which registers are to be stored and can contain any of the registers in the range R0 to R15. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 120.

The instruction is atomic.

Higher numbered registers are stored at higher memory addresses.

Registers are stored low register first.

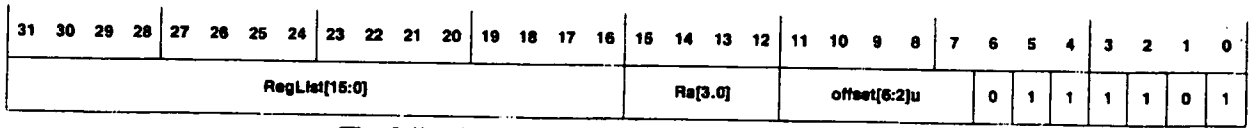
A `NullPointer` exception is thrown if any of the memory addresses are zero or within the MMU's defintion of null pointers, and the instruction terminates. Stores prior to the exception complete normally.

An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
stm (%r1, %r3, %r5, %r6), @(24, %r7)
```

**32-bit Encoding**




The following rules apply:

- The meaning of RegList[15:0] is.

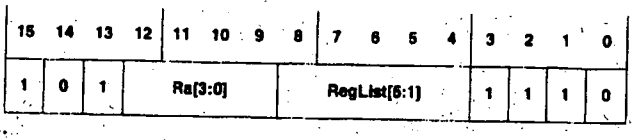
| RegList bit | register |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R2 |
| etc | |
| 14 | R14 |

| 15 | R15 |
|---|---|

- The offset can take values in the range 0, 4, ..., 116, 120. An offset of 124 is reserved.

**16-bit Encoding**

The following 16-bit encoding can be used:




The following rules apply:

- The bits in RegList[5:1] have the same meaning as in the 32-bit encoding:

| RegList bit | register |
|---|---|
| 1 | R1 |
| 2 | R2 |
| 3 | R3 |
| 4 | R4 |
| 5 | R5 |

# stm.u — Store multiple, user

**Instruction** `stm.u RegList, @(0, Ra)`

**Description** Store multiple user registers to memory

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

`stm.u` always stores the User Mode registers, irrespective of the processor's current operating mode.

The RegList operand specifies which registers are to be storedand can contain any of the registers in the range R0 to R15. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The instruction is atomic.

Higher numbered registers are stored at higher memory addresses.

Registers are stored low register first.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples** `stm.u {%r1, %r5, %r6}, @(0, %r7)`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RegList[15:0] | | | | | | | | | | | | | | | | Ra[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

The following rules apply:

- The meaning of RegList[15:0] is:

| RegList bit | register |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R2 |
| etc | |
| 14 | R14 |
| 15 | R15 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# stm.x — Store multiple, exchanged order

**Instruction** `stm.x RegList, @(offset, Ra)`

**Description** Store multiple registers to memory

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes**

`stm.x` loads the registers being used by the current processor operating mode.

The RegList operand specifies which registers are to be stored and can contain any of the registers in the range R0 to R15. Register Ra may be one of the registers specified in RegList. Refer to section Register Lists for details on RegList specifications.

The offset can take values in the range 0 to 120.

The instruction is atomic.

Higher numbered registers are stored at lower memory addresses.

Registers are stored high register first.

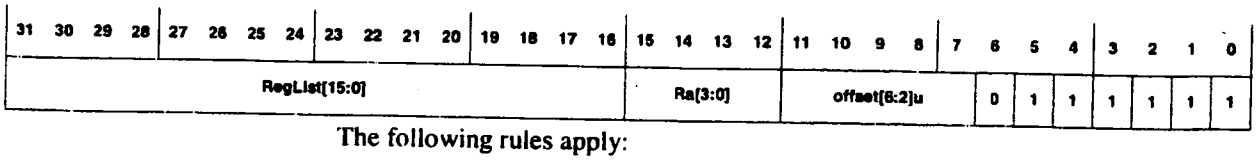
A `NullPointer` exception is thrown if any of the memory addresses are zero or within the MMU's defintion of null pointers, and the instruction terminates. Stores prior to the exception complete normally.

An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
stm.x {%r1, %r3, %r5, %r6}, @(120, %r7)
```

**32-bit Encoding**




The following rules apply:

- The meaning of RegList[15:0] is:

| RegList bit | register |
|---|---|
| 0 | R0 |
| 1 | R1 |
| 2 | R2 |
| etc | |
| 14 | R14 |

| 15 | R15 |
|---|---|

- The offset can take values in the range 0, 4, ..., 116, 120. An offset of 124 is reserved.

**16-bit Encoding**

There is no 16-bit encoding for this instruction.

# sub.c.r — Subtract, with carry, register

**Instruction** `sub.c.r Rd, Rs, Rt`

**Description** Subtract with carry

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs - Rt - FLAGS.C`

**Usage Notes** -

**Examples** `sub.c.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3.0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Rt[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sub.r — Subtract, register

**Instruction** `sub.r Rd, Rs, Rt`

**Description** Subtract

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs - Rt`

**Usage Notes** -

**Examples** `sub.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 1 1 1 1 | 1 1 1 1 | 0 0 0 1 | Rt[3:0] | 0 1 1 1 | 1 1 1 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd[3:0] | Rt[3:0] | 0 0 1 0 | 0 0 0 0 |

The following rules apply:

- Rd and Rs are the same register
- Rd=0 is reserved
- Rt=0 is reserved

# sub.x.h Subtract, exchanged order, high immediate

**Instruction** `sub.x.h Rd, Rs, #immediate`

**Description** Exchanged order subtract immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = #immediate - Rs`

**Usage Notes** Refer to section "h instructions - "h" encoding" for details of immediate encoding.

**Examples**

```
sub.x.h %r1, %r2, #0x12340000

sub.x.h %r1, %r2, #0x4321FFFF
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | #immediate[31:15]h | | | | 1 0 0 0 | 0 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# sub.x.i — Subtract, exchanged order, immediate

**Instruction** `sub.x.i Rd, Rs, #immediate`

**Description** Exchanged order subtract immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

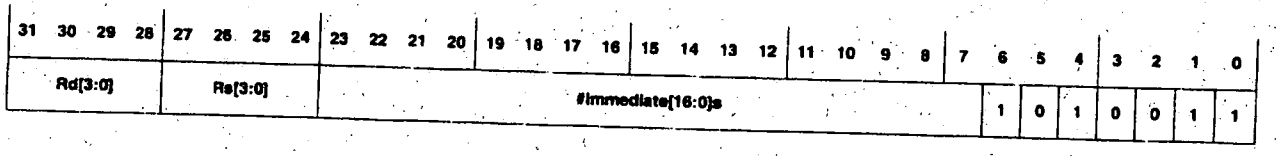
**Operation** `Rd = #immediate - Rs`

**Usage Notes** Refer to section "Instructions - 's' encoding" for details of immediate encoding.

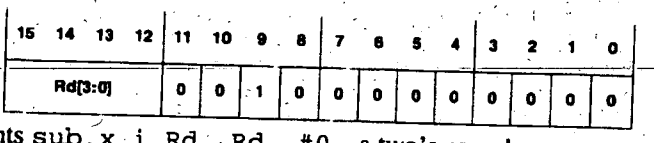
**Examples**

```
sub.x.i %r1, %r2, #0x00001234
sub.x.i %r1, %r2, #0xFFFF4321
```

**32-bit Encoding**




**16-bit Encoding** The following 16-bit encoding can be used:




This instruction implements `sub.x.i Rd, Rd, #0` – a two's complement or NEG (negate) instruction.

## SVC.i — Set if overflow clear, immediate

**Instruction** `svc.i Rd, Rs, #immediate`

**Description** Set register if overflow clear

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = (FLAGS.V == 0) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions - 'S' encoding" for details of immediate encoding.

**Examples**

```
svc.i %r1, %r2, #0x00000123
svc.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 | 1 | 1 | 1 | #immediate[12:0]s | | | 0 1 0 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## svc.r — Set if overflow clear, register

**Instruction** `svc.r Rd, Rs, Rt`

**Description** Set register if overflow clear

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.V == 0) ? Rs : Rt`

**Usage Notes** -

**Examples** `svc.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## svs.i — Set if overflow set, immediate

**Instruction** `svs.i Rd, Rs, #immediate`

**Description** Set register if overflow set

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = (FLAGS.V == 1) ? Rs : #immediate`

**Usage Notes** Refer to section "Instructions - "S" encoding" for details of immediate encoding.

**Examples**

```
svs.i %r1, %r2, #0x00000123
svs.i %r1, %r2, #0xFFFFF321
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd[3:0] | Rs[3:0] | 0 1 1 0 | #immediate[12:0]s | | | 0 1 0 | 1 1 1 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## SVS.r — Set if overflow set, register

**Instruction** `svs.r Rd, Rs, Rt`

**Description** Set register if overflow set

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (FLAGS.V == 1) ? Rs : Rt`

**Usage Notes** -

**Examples** `svs.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

## swap.r — Swap register with memory

**Instruction** `swap.r Rd, @(Rx, Ra)`

**Description** Swap register with memory

**Flags**

| | |
|---|---|
| **Z** | Set if the new Rd value is zero; cleared otherwise |
| **N** | Set if the new Rd value is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int32*)( 4*Rx + Ra ) <-> Rd`

**Usage Notes** The `swap.r` instruction performs an atomic swap between a register and a memory location.

Ra is interpreted as a byte address. Rx is interpreted as a four-byte offset from Ra.

A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `swap.r %r1, (%r2, %r3)`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | Ra[3:0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | Rx[3:0] | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# trap.i

## Trap, immediate

**Instruction** `trap.i #immediate`

**Description** Enter Supervisor Mode from User Mode

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** -

**Usage Notes** The `trap` instruction throws a `Trap` exception. This allows code executing in User Mode to call Supervisor Mode functions. The `trap` instruction is executed as a nop in the Supervisor, Interrupt and NMI modes.

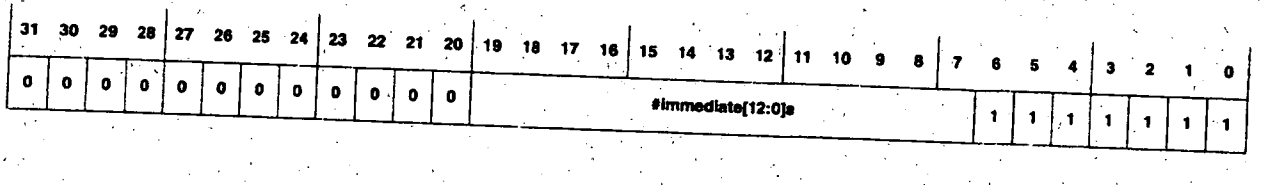
The immediate can take values in the range -4096 to +4095.

**Examples**

```
trap.i #0x00000123

trap..i #0xFFFFF321
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# trap.r

## Trap, register

**Instruction** `trap.r Rs`

**Description** Enter Supervisor Mode from User Mode

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes** The `trap` instruction throws a `Trap` exception. This allows code executing in User Mode to call Supervisor Mode functions. The `trap` instruction is executed as a `nop` in the Supervisor, Interrupt and NMI modes.

**Examples**

```
trap.r %r1
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# ver — Read XAP3 version number

**Instruction** `ver Rd`

**Description** Read the XAP3's version number

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = XAP3 version number`

**Usage Notes** This instruction reads the XAP3 version number. The format of this number is proprietary to Cambridge Consultants. Software applications may use this number to determine which features are implemented on the XAP3 processor and which require emulation.

**Examples** `ver %r1`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** There is no 16-bit encoding for this instruction.

# xor.h — Exclusive-or, high immediate

**Instruction** `xor.h Rd, Rs, #immediate`

**Description** Exclusive-OR with high immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

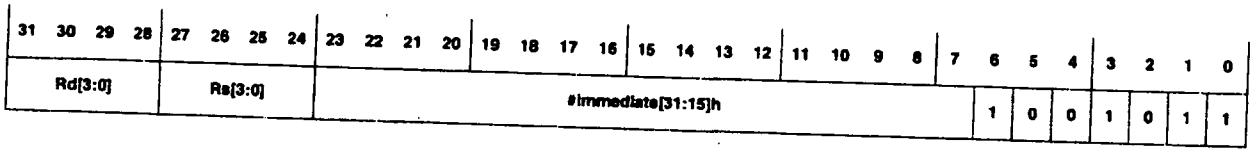
**Operation** `Rd = Rs ^ #immediate`

**Usage Notes** Refer to section "h instructions - h" encoding" for details of immediate encoding.

**Examples**

```
xor.h %r13, %r5, #0x12340000
xor.h %r13, %r5, #0x4321FFFF
```

**32-bit Encoding**




**16-bit Encoding** There is no 16-bit encoding for this instruction.

# xor.i — Exclusive-or, immediate

**Instruction** `xor.i Rd, Rs, #immediate`

**Description** Exclusive-OR with immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

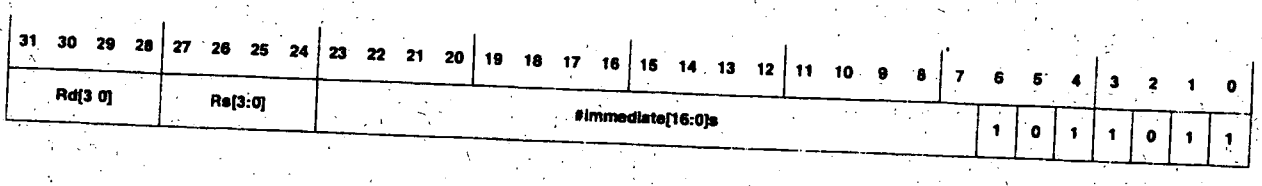
**Operation** `Rd = Rs ^ #immediate`

**Usage Notes** Refer to section "Instructions – 's' encoding" for details of immediate encoding.

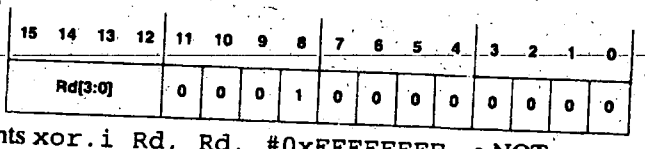
**Examples**

```
xor.i %r1, %r2, #0x00001234
xor.i %r1, %r2, #0xFFFF4321
```

**32-bit Encoding**




**16-bit Encoding** The following 16-bit encoding can be used:




This instruction implements `xor.i Rd, Rd, #0xFFFFFFFF` – a NOT (one's complement) instruction.

## xor.r — Exclusive-or, register

**Instruction** `xor.r Rd, Rs, Rt`

**Description** Exclusive-OR

**Flags**

| | |
|---|---|
| **Z** | Set if the result is positive; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = Rs ^ Rt`

**Usage Notes** -

**Examples** `xor.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3.0] | | | | Rs[3:0] | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | Rt[3:0] | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**16-bit Encoding** The following 16-bit encoding can be used:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd[3:0] | | | | Rt[3:0] | | | | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

The following rules apply:

- Rd and Rs are the same register
- Rd=0 is reserved
- Rt=0 is reserved

# Index

*A*

Addressing Modes ..... 49
  Displacement ..... 50
  Indexed addressing ..... 50
Aliased Instructions ..... 79
Application Models ..... 39
Assembler Syntax ..... 41

*C*

Calling Convention ..... 39

*D*

Data Alignment ..... 36

*E*

Endianness ..... 36
Exceptions ..... 25
  Break ..... 30
  DivideByZero ..... 31
  IllegalCodeAddress ..... 33
  IllegalDataAccess ..... 32
  IllegalInstruction ..... 31
  IllegalInstructionAccess ..... 32
  IllegalStackAddress ..... 33
  in the Privileged Modes ..... 26
  in User Mode ..... 26
  NullPointer ..... 31
  Returning from ..... 28
  SingleStep ..... 30
  Trap ..... 30
  UnknownInstruction ..... 30

*I*

Immediate Encoding ..... 44
Interrupt vector table ..... 28
Interrupts ..... 25, 27
  Nested ..... 28
  Returning from ..... 28

*M*

Memory Architecture ..... 24

*N*

Non-Maskable Interrupts ..... 28

*P*

Pipeline ..... 24
Processor Modes ..... 16
Processor States ..... 15

*R*

Register Lists ..... 42
Registers
  64-bit ..... 20
  Break Enable ..... 23
  Breakpoint ..... 23
  Flags ..... 22
  Global Pointer ..... 23
  Interrupt Vector Table Base ..... 23
  Normal ..... 18
  Program Counter ..... 20
  Saved Flags ..... 23
  Special ..... 21
Reset ..... 25

*S*

Stack ..... 25

*X*

XAP Family
  XAP1 ..... 9
  XAP2 ..... 9
  XAP3 ..... 7
xIDE ..... 33

Cambridge Consultants

# XAP4 Specification

Prepared by
**AGM**

Reviewed by
**KLS, DWH2**

Secretary
**CTS**

**Cambridge Consultants Ltd**
Science Park
Milton Road
Cambridge
England CB4 0DW

Tel +44 (0)1223 420024
Fax +44 (0)1223 423373
info@CambridgeConsultants.com
www.CambridgeConsultants.com

C7432-S-001 v2.0
29 November 2005

**Subject:** XAP4 Specification

## Distribution

Codemist

Cambridge Consultants

Project Master file

## Cambridge Consultants Ltd – Project Team

Project Reviewer

Project Manager

Technical Contributors

Project Secretary

# Table of Contents

**1 Introduction** 7
1.1 Release History 8
1.2 Related Documents 8

**2 xIDE for XAP4** 9
2.1 xIDE Integrated Development Environment 9
2.2 Software Simulator 9
2.3 Hardware Emulator 9
2.4 Compiler, Assembler, Linker 9
2.5 Assembler Syntax 11
2.5.1 Syntax Strategy 11
2.5.2 Syntax Rules 11
2.5.2.1 General 11
2.5.2.2 Names 12
2.5.2.3 Directives and Labels 12
2.5.2.4 Numbers 13
2.5.2.5 Instruction Format 13
2.5.2.6 Instruction Mnemonics 13
2.5.2.7 Registers 14
2.5.2.8 Accumulators 15
2.5.2.9 Immediates 16
2.5.2.10 Addresses 16
2.5.3 Syntax Examples 16

**3 Key Hardware Features** 18
3.1 CPU 18
3.2 Memory 19
3.3 Hardware Implementation 19
3.4 Debug Support 19

**4 Processor Architecture** 20

**5 Programmer's Model** 22
5.1 Processor Mode 22
5.2 Processor State 24
5.3 Run State 24
5.4 Normal Registers 24
5.5 Breakpoint Registers 25
5.6 Special Registers 26
5.7 Flags 27
5.8 Memory Model 28
5.9 Reset State 29
5.10 Calling Convention 29
5.11 Application Models 30
5.11.1 Statically Linked 30
5.11.2 Dynamically Loaded 30

**6 Interrupts and Exceptions** 31
6.1 Overview 31
6.2 Exception Processing 32

6.3 Interrupt Processing 33
6.4 Returning from Interrupts and Exceptions 34
6.5 Nested Interrupts 35
6.6 Interrupt Vector Table 35
6.7 Exceptions 36
6.7.1 Reset 36
6.7.2 NMI 36
6.7.3 Trap 36
6.7.4 SingleStep 37
6.7.5 Break 37
6.7.6 UnknownInstruction 37
6.7.7 DivideByZero 38
6.7.8 NullPointer 38
6.7.9 IllegalInstruction 39
6.7.10 IllegalInstructionAccess and IllegalDataAccess 39
6.7.11 IllegalCodeAddress 40

**7 Addressing Modes** **41**
7.1 Register, Register, Immediate 41
7.2 Register, Register, Register 41
7.3 Load, Store 42
7.3.1 Displacement Addressing 42
7.3.2 PC-Relative Addressing 42
7.3.3 Indexed Addressing 42

**8 Instruction Set** **43**
8.1 Notation Conventions 43
Structure of 16-bit Instructions 44
Structure of 32-bit Instructions 45
8.4 Base Instructions 47
8.5 Aliased Instructions 53
8.6 Flag Usage Conventions 53
8.7 Interpretation of Immediates 54
8.7.1 Immediates 54
8.7.2 Addresses 55
8.8 Instruction Opcodes 56
8.8.1 OP = 0 – 3, 16-bit instructions 57
8.8.2 OP = 4, 16-bit instructions 58
8.8.3 OP = 5, 16-bit instructions 59
8.8.3.1 OP = 5, Branches 59
8.8.3.2 OP = 5, Push, Pop 59
8.8.3.3 OP = 5, 16-bit Shifts 60
8.8.4 OP = 6, 16-bit instructions 61
8.8.5 OP = 7, 32-bit RRI instructions 63
8.8.6 OP = 7, 32-bit RRRI instructions 66
8.8.7 OP = 7, 32-bit RRR instructions 67
8.9 Instruction Formats 69
8.9.1 Unconditional Branch Instructions 69
8.9.1.1 Absolute 69
8.9.1.2 PC Relative 69
8.9.1.3 Register 70
8.9.2 Conditional Branch Instructions 71
8.9.3 Load and Store Instructions 73

8.9.3.1 Displacement Addressing (offset, Ra) – ld.*.i, st.*.i 73
8.9.3.2 Indexed Addressing (Rx, Ra) – ld.*.r, st.*.r 74
8.9.3.3 PC-Relative Addressing (offset) – ld.*.p, st.*.p 75
8.9.3.4 Store Zero Instructions 76
8.9.3.5 swap.i Instruction 77
8.9.4 push, pop, pop.ret Instructions 78
8.9.4.1 push 79
8.9.4.1.1 push example 1 80
8.9.4.1.2 push example 2 80
8.9.4.2 pop 81
8.9.4.2.1 pop example 1 81
8.9.4.2.2 pop example 2 82
8.9.4.3 pop.ret 83
8.9.4.3.1 pop.ret example 1 83
8.9.4.3.2 pop.ret example 2 84
8.9.5 mov Instructions 85
8.9.5.1 mov.p – PC-relative Offset 85
8.9.5.2 mov.i - Immediate 85
8.9.5.3 mov.r – Register 86
8.9.5.4 mov.32.r – Register 86
8.9.6 ALU Instructions 87
8.9.6.1 Logic Instructions 87
8.9.6.2 Arithmetic Instructions 87
8.9.6.3 Multiply Instructions 88
8.9.6.4 Divide/Remainder Instructions 88
8.9.6.5 Reg-Reg-Immediate Format 90
8.9.6.6 Reg-Reg-Reg Format 91
8.9.7 Compare Instructions 92
8.9.7.1 Compare Assembler Format 92
8.9.7.2 Compare Flags 93
8.9.7.3 Compare Immediate 93
8.9.7.4 Compare Register 94
8.9.8 Shift and Rotate Instructions 95
8.9.8.1 16-bit Shift and Rotate Instructions 95
8.9.8.1.1 Reg-Reg-Immediate 95
8.9.8.1.2 Reg-Reg-Reg 96
8.9.8.2 32-bit Shift and Rotate Instructions 96
8.9.8.2.1 Reg-Reg-Immediate 97
8.9.8.2.2 Reg-Reg-Reg 97
8.9.9 Block Instructions 98
8.9.9.1 blkcp.r – Block Copy 98
8.9.9.2 blkcps.r – Block Copy String 99
8.9.9.3 blkst.r - Block Store 99
8.9.9.4 blkst.8.r – Block Store (8-bit) 101
8.9.10 DSP Instructions 102
8.9.10.1 mult.sh.r – Multiply then shift right 102
8.9.11 Special Register Instructions 103
8.9.12 System Instructions 104
8.9.12.1 16-bit System Instructions 104
8.9.12.2 32-bit System Instructions 105

**9 SIF Serial InterFace** **106**
9.1 SIF ASIC Block Diagram 106

9.2 xSIF and iSIF usage modes 107
9.3 SIF ASIC Modes 108
9.3.1 SIF Modes 108
9.3.1.1 SIF Mode = Normal 109
9.3.1.2 SIF Mode = Command 109
9.3.2 XAP4 SIF Processor Modes 110
9.3.2.1 XAP4 SIF Processor Mode = Normal 110
9.3.2.2 XAP4 SIF Processor Mode = Debug 110
9.4 When are SIF Operations executed ? 112
9.4.1 SIF Mode = Command 112
9.4.2 SIF Mode = Normal 112
9.4.2.1 Immediate SIF Handshake 113
9.4.2.2 Delayed SIF Handshake 113
9.4.2.3 Incomplete SIF Handshake (ending in SIF Lockup) 113
9.4.2.4 SIF Cancel (to clear SIF Lockup) 114
9.5 SIF Shift Register 115
9.5.1 Address Field 115
9.5.2 Control Field 115
9.5.3 Data Field 116
9.5.4 Status Field 116
9.5.5 Shift Register Bit Definitions 117
9.6 SIF Instructions 120
9.6.1 SIF Instruction Types 120
9.6.2 SIF Instruction Table 121
9.6.3 XAP4 Status Register 126
9.7 SIF Error Processing 128
9.7.1 SIF Parity Bits 128
9.7.1.1 PARITY_AC 128
9.7.1.2 PARITY_D 128
9.7.1.3 PARITY_S 128
9.7.2 SIF Error Reporting in Status Field 129
9.7.2.1 SIF Error Codes 129
9.7.2.2 Processor Error Codes 129
9.8 SIF Resets 130

# 1 Introduction

This document is the specification for the XAP4, CCL's new 16-bit ASIC processor. The XAP4 uses the same load-store philosophy, instruction set and programmer's model as XAP3 (CCL's 32-bit ASIC Processor). XAP4 is a Von Neumann processor optimised for low-power and low gate-count. It supports a flat 64kB memory space using little-endian byte addressing.

The associated software toolkit is 'xIDE for XAP4'. xIDE is CCL's generic Integrated Development Environment and runs on most computing platforms.

The XAP4 instruction set and toolkit have been developed before the hardware. The instruction set has been optimised to give good code density, leading to low power consumption and low silicon cost. The compile chain is based on the GNU tools, GCC C compiler and binutils assembler and linker.

XAP4 is aimed at all CMOS processes from 0.7 to 0.09 µm. Clock speeds depend upon the specific hardware implementation and are a trade-off with power consumption. It is expected to run at clock speeds from 10 kHz to 100MHz. The small hardware implementation contains about 8k gates with a typical memory manager.

XAP4 is intended to be used with on-chip RAM (normally 16-bit synchronous) and with on or off-chip Flash (normally 16-bit asynchronous). The default XAP4_mmu supports 16-bit memories.

XAP4 has 16-bit and 32-bit instructions. These can be freely mixed in any pattern. All instructions are 16-bit aligned. This means that all 16-bit instructions are aligned. 32-bit instructions can be unaligned (to any even address).

XAP4 supports 4 operating modes (User, Supervisor, Interrupt, NMI). XAP4 has a load-store architecture. XAP4 includes powerful debug features including the SIF serial interface. The XAP4 SIF supports up to 8 on-chip processors and parity-checking.

XAP4 has three banks of registers. The Normal Registers bank contains 8 general purpose 16-bit registers. Some of these are shadowed in Supervisor, Interrupt and NMI modes. There is one Breakpoint Register which is accessible from Interrupt mode. The Special Registers bank allows access to internal processor registers including the shadow registers and flags.

XAP4 can be used with up to 64kB of memory. This is accessed with byte addresses (i.e. 16-bit address) and assumes Little Endian byte ordering for all words longer than 8-bits. All data memory is unaligned (8 and 16-bit data can have any byte address).

Each program can access 64kB of memory which can be split as :

- 64kB of code/constants
- 64kB of global variables
- Up to 64kB of heap
- Up to 64kB of stack

- Up to 64kB per function

Programs are compiled to be relocatable by using PC-relative addressing for code, constants and global variables. Programs can be loaded immediately next to each other in memory. There are no memory segments.

## 1.1 Release History

| *Version* | *Date* | *Comments* |
|---|---|---|
| | April 2005 | Copied from C7245-S-001 v2.28 (XAP3 Specification) |
| | May 2005 | First internal release |
| 0.1 | 8 June 2005 | First DocMan Release as C7432-S-001 |
| 0.4 | 17 June 2005 | Last version used by Saajan Chana |
| 0.8 | 7 July 2005 | Modified Z-flag rules for carry versions of add, subtract and compare instructions. |
| 0.9 | 20 July 2005 | Added div.*, rem.*, divrem.* instructions. |
| 0.10 | 27 July 2005 | Added 3 Block instructions.<br>Allowed 32-bit accumulators to start at any register.<br>Changed shift.32* to 32-bit instructions. |
| 0.12 | 1 Aug 2005 | Changed R0 to be a real register (instead of constant 0) |
| 1.0 | 1 Aug 2005 | Added stz.*.r instructions. |
| 1.2 | 15 Aug 2005 | Added ld.*.p, st.*.p, stz.*.p, mov.32.5 instructions.<br>Changed some 16-bit instruction codes. |
| 1.4 | 8 Nov 2005 | Added iSIF. Removed 4 SIF Counters. Added 16bit stz.i.<br>Changed push, pop, pop.ret reglist decode. |
| 1.5 | 21 Nov 2005 | Removed 16-bit version of bsr.p instruction.<br>Changed Flag settings for mult.i, mult.r, pop, pop.ret.<br>Added SIF Error Code for 'Cancelled iSIF Instruction'. |
| 1.6 | 28 Nov 2005 | Updated clock cycles per instruction and other errors. |
| 2.0 | 29 Nov 2005 | Small changes to calling conventions in sections 5 and 6. |

## 1.2 Related Documents

| *Reference* | *Title* | *Author* |
|---|---|---|
| C4862-TM-005 | SIF Definition | AGM |
| C7245-S-001 | XAP3 Specification | AGM |
| C7245-UM-002 | XAP3 Programmer's Manual | MDC |
| C7432-UM-002 | XAP4 Programmer's Manual | MDC |
| C7432-UM-003 | XAP4 BinUtils Manual | KLS |
| C7432-UM-004 | XAP4 GCC Compiler Manual | NEJ |
| C7432-UM-011 | XAP4a Hardware Reference Manual | AGM, DWH2 |

# 2 xIDE for XAP4

## 2.1 xIDE Integrated Development Environment

The software Integrated Development Environment (IDE) for XAP4 is based on CCL's xIDE. xIDE allows plugins to be created that add support for new processors, build systems and debuggers.

xIDE uses Qt from Trolltech and can be compiled for Windows PC, Apple Mac OS X, and X11 platforms (including Sun and Linux).

The following documents describe xIDE for XAP4:

| | |
|---|---|
| C7066-TM-001 | xIDE User Manual |
| C7066-TM-003 | xIDE Python Object Model |
| C7066-TM-017 | XAP4 Plugin |

## 2.2 Software Simulator

The software simulator is part of the XAP4 plugin for xIDE.

The Simulator allows Python models of hardware to be added to simulate the effect of hardware registers and interrupts.

## 2.3 Hardware Emulator

The Emulator is part of the XAP4 plugin for xIDE and provides a connection to the target board via a SIF physical interface. Cambridge Consultants sells PC test hardware and software drivers for the SIF. Other physical interfaces to the XAP4 Emulator (e.g SPI, I2C, JTAG, USB, RS232) can be developed if needed. The default target emulator board is xEMU, Cambridge Consultants' standard Digital Emulator (C6707-DL-002).

## 2.4 Compiler, Assembler, Linker

The C compiler (xap4-gcc) is based on GCC. Using GCC also allows access to C++ if wanted in the future.

The assembler and linker are based on GNU binutils.

GCC and Binutils are released under the GNU General Public License. Binutils allows us to take advantage of standard ELF object files with embedded Stabs debug information.

The word sizes that the C compiler assumes are as follows :

| *C compiler Type* | *No of bits* |
|---|---|
| Char | 8 |
| Short | 16 |
| Long | 32 |
| Int | 16 |
| Pointer | 16 |

A XAP4 word is defined to be 16-bit.

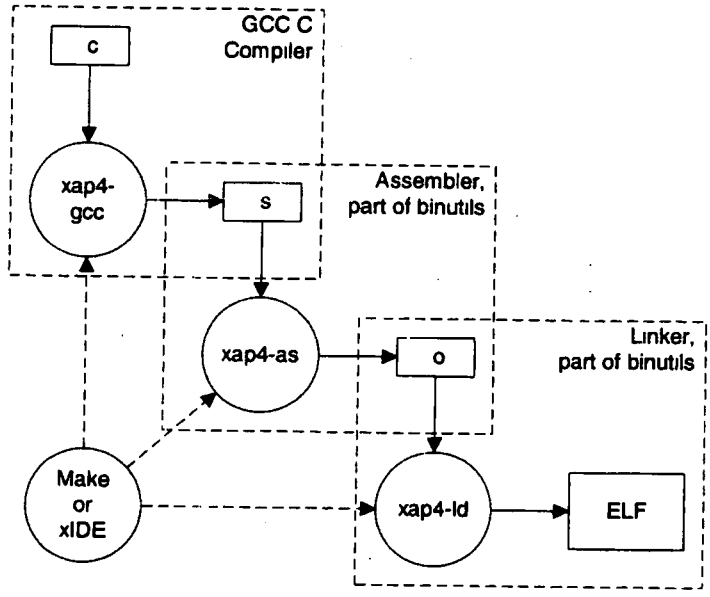
There is no hardware support for the data types; long, float, double.

There is xap4-gcc/library software support for long, float and double. Long is treated as 32-bit. Float and Double are treated identically as 64-bit words. They are stored in memory as structures and are unaligned (can have any byte address). The format used to store floats and doubles is a property of the compiler and library, not of the processor itself. The format used by the compiler and library for floating point variables is IEEE 754 Standard, implemented as little-endian (bit order within words and word order within the structure). This is consistent with the hardware support for 8-bit and 16-bit variables in the processor itself.

Addresses are computed in two ways: PC-relative and absolute. The compiler will generate PC-relative addresses for all code and data objects (constants and variables).

The instruction set contains 16-bit and 32-bit branch instructions. The compiler generates instruction mnemonics (in the .s file) and the assembler decides whether the instructions will be 16 or 32-bit. All branch mnemonics generated by the compiler are PC relative branches. Conditional branches generated by the compiler will be to locations within the same function; the assembler will compact most of these branches to 16-bit instructions. Unconditional branches within a source file (module) will be either 16 or 32-bit instructions (decided by the assembler). Unconditional branches between modules will be 32-bit instructions and will be resolved by the linker. By having all inter-module branches as 32-bit instructions, we avoid the need to perform relaxation in the linker, which makes the process more efficient. Future linkers may perform relaxation, so that some unconditional branches between modules could be implemented as 16-bit instructions.

The diagram below shows the relationship between xap4-gcc and Binutils.




## 2.5 Assembler Syntax

### *2.5.1 Syntax Strategy*

The assembler syntax for XAP4 is the same style as for XAP3. Most of the mnemonics are identical. Immediates are represented as 16-bit (whereas XAP3 is 32-bit). The choice of syntax rules for the assembler language used in .s files is driven by the following rules (most significant first) :

- No name-space pollution (e.g. for source files written in C)
- Occam's Razor – provide only one way to do things
- Binutils – follow the conventions normally used
- Case-sensitive – all reserved words will be lower-case
- C++ style syntax wherever possible (e.g. comments and numbers)
- XAP2 assembler – follow the conventions used there

### *2.5.2 Syntax Rules*

#### 2.5.2.1 General

The assembler is line-based. Instructions are terminated by end-of-line.

Spaces and tabs are treated as white-space that delimit tokens.

All assembler is interpreted as case-sensitive, except hex numbers.

All reserved words are lower-case (instruction mnemonics, registers, number prefixes ..)

Line Comments are from // to end-of-line.

Block comments (can be across 1 or more lines) are from /* to */.

#### 2.5.2.2 Names

Names from C source will not be modified. Thus labels and variable names will be exactly the same as in the original C. However, local static symbols will have an assembler name of the format :

- basefilename.symbolname // e.g module.function

Hardware names will be prefixed by %. Normal registers will be referred to as %r0 to %r7. 32-bit accumulators are formed from an adjacent pair of registers. Accumulators are referenced with the lower register of the pair, e.g :

- %r1 references {R2, R1}
- %r2 references {R3, R2}

The accumulators are only used by the following instructions :

- shift*.32*
- rotatel.32*
- mult.32*
- div.32*
- rem.32*
- divrem.32*

The Special Registers and Breakpoint Registers will be referred to by the names defined in sections 5.6 and 5.5.

#### 2.5.2.3 Directives and Labels

Directives start with period (".") as the first non-blank character of the line. For example:

- `.org`
- `.file`
- `.text`

Labels start from the beginning of a line. They are appended with colon (":"). For example:

- `label_35A:`

#### 2.5.2.4 Numbers

Number formats used in XAP4 Assembler are :

| *Prefix* | *Example* | *Description* | *High bits up to Bit 15* | *Comment* |
|---|---|---|---|---|
| 0x | 0x12ab<br>0x12AB | Hex | Set to 0 | |
| | 103 | Signed Decimal | Sign Extended | Default |
| | -7521 | Signed Decimal | Sign Extended | Default |
| | 58297 | Unsigned Decimal | Set to 0 | |
| 0b | 0b11001 | Binary | Set to 0 | |

Positive signed decimal and unsigned decimal numbers look the same. They will be interpreted as required for that particular instruction (which may expect a signed or unsigned number). The Assembler will reject numbers that are of invalid type or are out of range for that particular instruction. More detail in the later section on Immediates.

#### 2.5.2.5 Instruction Format

Instructions can take several forms :

- mnemonic label // Conditional and Unconditional Branches
- mnemonic %registers
- mnemonic %registers, #immediates
- mnemonic %registers, @address

The convention is to have a tab between the mnemonic and the first argument. This should not be required by the binutils assembler, but should be generated by the C compiler.

Instructions must explicitly list every argument for each instruction. e.g. :

- ld.i %r2, @(0, %r4) // NOT ld.i %r2, @%r4

#### 2.5.2.6 Instruction Mnemonics

The mnemonic is a string in one of the following formats :

- Base
- Base.Parameters
- Base.Type
- Base.Parameters.Type
- Base.Parameters.Type.Size

Examples of Base are :

- add
- mov
- ld
- bra

Examples of Parameter are :

- c // with carry
- x // exchange order (= reverse order)
- z // zero-extend
- s // signed
- u // unsigned
- 8 // 8-bit operation
- 32 // 32-bit operation

Examples of Type are :

- r // register
- i // immediate
- a // absolute address
- p // PC-relative address

Examples of Size are :

- 2 // 2byte, 16-bit instruction
- 4 // 4byte, 32-bit instruction

Instruction mnemonics do not normally include Size. It is only used when the xap4-gcc compiler wants to force the binutils assembler/linker to use a particular instruction size. This is sometimes needed for branch instructions.

Here are some alternative versions of the add instruction :

- add.r // Add Rt to Rs and store result in Rd
- add.c.r // Add Rt to Rs with carry and store result in Rd
- add.i // Add Immediate to Rs and store result in Rd

#### 2.5.2.7 Registers

The 8 registers are referred to as %r0 to %r7.

There can be several Registers in each instruction. They will be in the following sequence from left to right :

- Rd // Destination Register
- Rs // Primary Source Register
- Rt // Secondary Source Register
- Rx // Index Address Register (scaled by 2 or 1 for 16 and 8 bit data)
- Ra // Base Address Register (address in bytes)

Each Register will be separated by, and prefixed by %.

Some instructions (push, pop) require a list of registers as an argument. The number of registers can vary. Such lists are enclosed in { }. e.g. :

- push {%r3, %r6}, #6

Within such register lists, registers can be specified in ranges as follows :

- %r3-%r6 // The first register must be lower than the second.

This enables more compact Assembler code (from the C compiler and Disassembler).

Within the register list, the same register must not be specified more than once :

- {%r3-%r5, %r4} // Not valid because %r4 has been specified twice
- {%r3, %r5-%r6} // This is a valid register list

#### 2.5.2.8 Accumulators

Some instructions use 2 adjacent registers grouped together as a single 32-bit accumulator. XAP4 can use any adjacent register pair as a 32-bit accumulator. This assumes little-endian ordering and the accumulator is referenced by the lower register. The 32-bit instructions (mult.32, div.32, shift.32, rotate.32) will not be used with R7 (Stack Pointer), so none of the Accumulators include R7.

Valid Accumulators are %r0 - %r5 :

| Assembler Reference | Registers | Accumulator[31:16] | Accumulator[15:0] |
|---|---|---|---|
| %r0 | {R1, R0} | R1 | R0 |
| %r1 | {R2, R1} | R2 | R1 |
| %r2 | {R3, R2} | R3 | R2 |
| %r3 | {R4, R3} | R4 | R3 |
| %r4 | {R5, R4} | R5 | R4 |
| %r5 | {R6, R5} | R6 | R5 |

#### 2.5.2.9 Immediates

There can be several immediates in each instruction. Each Immediate will be separated by ','. Immediates outside '( )' are prefixed by #. For example:

```
shiftl.i %r4, %r6, #6
```

Immediate values in the assembly files are treated as 16-bit numbers. It is the responsibility of the assembler to decide whether the immediate value is compatible with the values permitted by the instruction mnemonics.

The assembler will sign-extend positive and negative decimal numbers, but not hex or binary. The assembler will then calculate whether the resultant 16-bit number is in a valid range for the Immediate field for that particular instruction. e.g :

```
add.i   %r1, %r2, #0x1234       // valid positive immediate
add.i   %r1, %r2, #35           // valid positive immediate
add.i   %r1, %r2, #-356         // valid negative immediate
add.i   %r1, %r2, #0xF234       // valid negative immediate
add.i   %r1, %r2, #0x111234     // invalid (immediate > 16 bit)
add.i   %r1, %r2, #0xABCD1234   // invalid (immediate > 16 bit)
add.i   %r1, %r2, #-100000      // invalid (immediate > 16 bit)
```

#### 2.5.2.10 Addresses

There can only be one Address in each instruction. This will be prefixed by @.

Address indirection is specified as one of :

- @(offset, Ra) @(-56, %r7) // Ra = byte address
- @(offset) @(125) // Used for PC-Relative addressing
- @(Rx, Ra) @(%r4, %r7) // Rx = scaled addr, Ra = byte addr

### *2.5.3 Syntax Examples*

Here are some valid instructions in XAP4 Assembler :

```
ld.i    %r2, @(1256, %r0)   // Load value at specified address into R2
ld.p    %r3, @(156)         // Load value at (PC + 156) into R3
st.r    %r5, @(%r3, %r6)    // Store register R5 at specified address
add.r   %r4, %r3, %r5       // Add R5 to R3 and store in R4
add.c.r %r7, %r2, %r4           /* Start of Block comment
foo bar
        End of Block Comment */ add.r  %r6, %r3, %r1 // this add.r is executed
or.i    %r3, %r7, #-563     // OR –563 with R7 and store in R3
and.i   %r1, %r5, #0xABcd   // AND hex ABCD with R5 and store in R1
```

```
xor.i    %r1, %r5, #0x1230      // XOR 0x1230 with R5 and store in R1
mult.r %r6, %r4, %r2            // 16x16 unsigned multiply of R4 with R2.
                                // 16-bit result stored in R6.
mult.32s.r %r5, %r4, %r2        // 16x16 signed multiply of R4 with R2.
                                // 32-bit result stored in accumulator {R6, R5}.
```

# 3 Key Hardware Features

## 3.1 CPU

16-bit internal architecture.

Unified linear 16-bit address space (64 kB).

Each program can use up to 64 kB (for code + constants + global variables + stack + heap). Instructions are stored on a 2-byte boundary. These can be positioned anywhere in memory. Separate programs can be loaded immediately next to each other. There are no memory segments to observe.

Byte addressing. All memory accesses assume Little-Endian byte ordering. If a 32-bit word (formed in software from 2 16-bit words) is at address 0x100, then the bits can be found at the following addresses :

- word[7:0] = address 0x100
- word[15:8] = address 0x101
- word[23:16] = address 0x102
- word[31:24] = address 0x103

If a 16-bit word is stored at address 0x205 (i.e unaligned), the bits can be found at the following addresses :

- word[7:0] = address 0x205
- word[15:8] = address 0x206

32-bit and 16-bit instructions are aligned on 16-bit address boundaries. i.e 16-bit instructions are aligned. 32-bit instructions can be aligned or unaligned (but they always have an even address).

The XAP4 hardware allows all data (8-bit, 16-bit, arrays and structures) to be unaligned (can have any byte address). However, the xap4-gcc compiler starts all data objects (except chars) on even addresses. Chars can be started on odd addresses. Groups of adjacent chars will be packed within a structure or array. All 16-bit words are started on even addresses (on their own or within structures or arrays). The xap4-gcc compiler does this because such code is faster and uses less power. The expense is that the data can require a little more memory than a fully packed system would.

All addresses and pointers are stored in registers as full 16-bit byte addresses. All address comparisons are done on full 16-bit byte addresses.

There are 3 banks of up to 8 registers. All registers are 16-bit :

- Normal Registers — see section 5.4
- Breakpoint Registers — see section 5.5

- Special Registers see section 5.6

The only instruction that can update a Special Register is movr2s. This can only be used by the privileged Supervisor, Interrupt and NMI modes. Similarly the movr2b instruction allows the privileged modes to update the breakpoint registers. All other processing is performed on the Normal registers.

Aimed at fixed point 16-bit processing. Support for 8-bit and 32-bit operations

Stack is 16 bits wide. Stack is word aligned (stack pointer address is always a multiple of 2). Stack Pointer address changes by ±2 (byte addressing). Stack grows downwards (standard for GCC).

Reasonable code density.

Good support for C compilation (suitable for GCC).

Four processor modes; User, Supervisor, Interrupt, NMI.

## 3.2 Memory

The XAP4 can be used with any combination of 8 and 16-bit synchronous and asynchronous memories.

The typical configuration will be to have internal 16-bit synchronous memories (RAM and registers) and external 8 or 16-bit asynchronous memories (parallel Flash).

## 3.3 Hardware Implementation

Clock speeds of up to 100MHz on 0.18 µm CMOS

Estimated size is 8k gates (including MMU).

Optimised for low cost (gate count) and low power

## 3.4 Debug Support

The SIF is described in section 9.

SingleStep and BreakExceptions are described in sections 6.7.4 and 6.7.5.

The breakpoint registers are described in section 5.5.

# 4 Processor Architecture

There are two versions of the XAP4 processor :

| *Processor* | *Features* | *Documentation* |
|---|---|---|
| XAP4a | Smallest possible XAP4<br>Slower<br>Lower power | C7432-TM-001 ??? |
| XAP4b | Faster arithmetic<br>Larger<br>Higher power | C7432-TM-002 ??? |

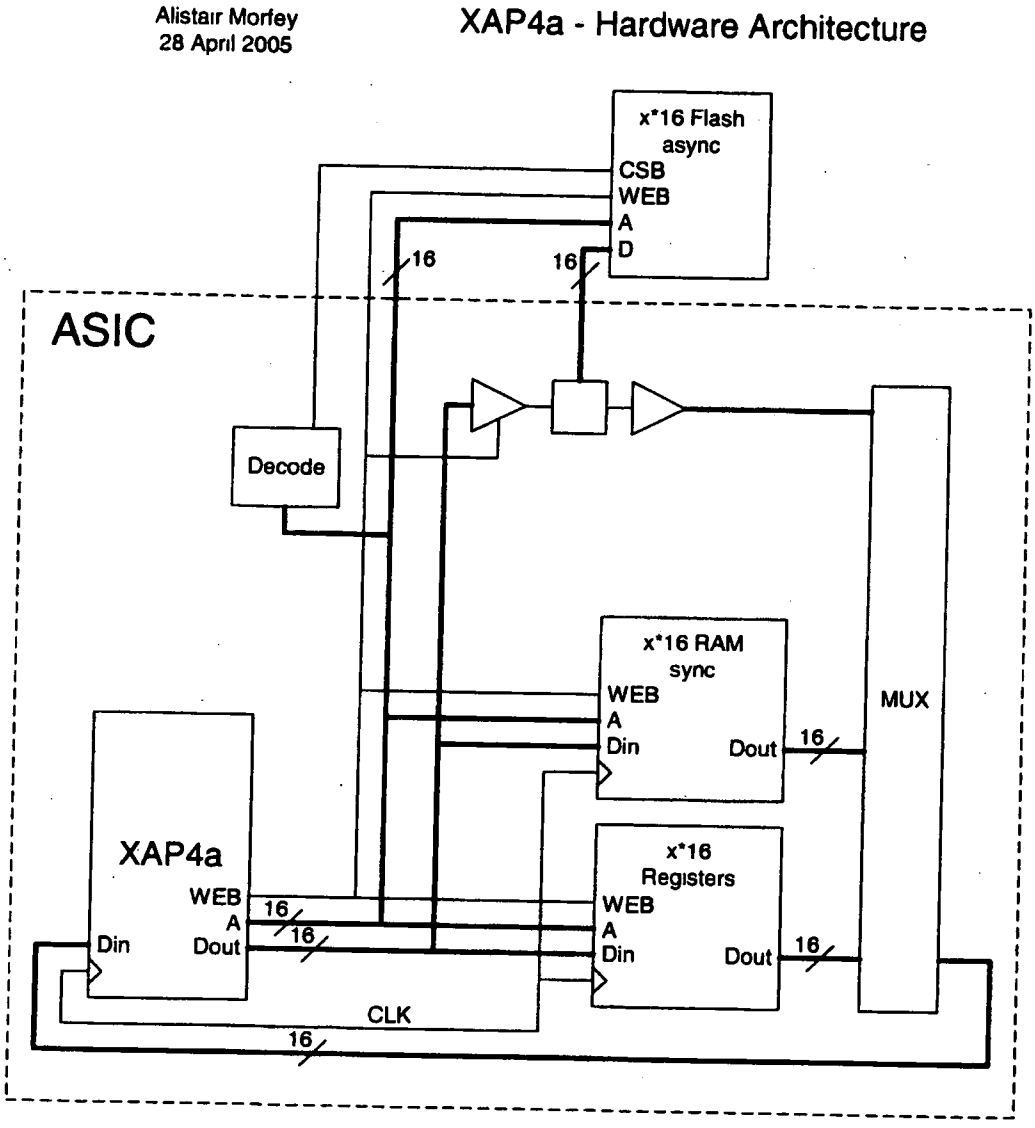
The default XAP4 architecture consists of:

- xap4_core – fixed and not changed from one ASIC to another
- xap4_debug – debug support to a serial interface (e.g SIF, SPI, $I^2C$, JTAG etc)
- xap4_mmu – Memory manager unit, likely to be changed from one ASIC to another

The debug serial interface normally used is SIF. This is a patented non-invasive serial interface developed by Cambridge Consultants. It is described in more detail at the end of this specification and in C4862-TM-005. The SIF is outside the XAP4. A single SIF can support up to 8 XAP4s (or a mixture of processors). Each processor can have its own address space (up to 64kB each), or can use shared address spaces.

**Subject:** XAP4 Specification

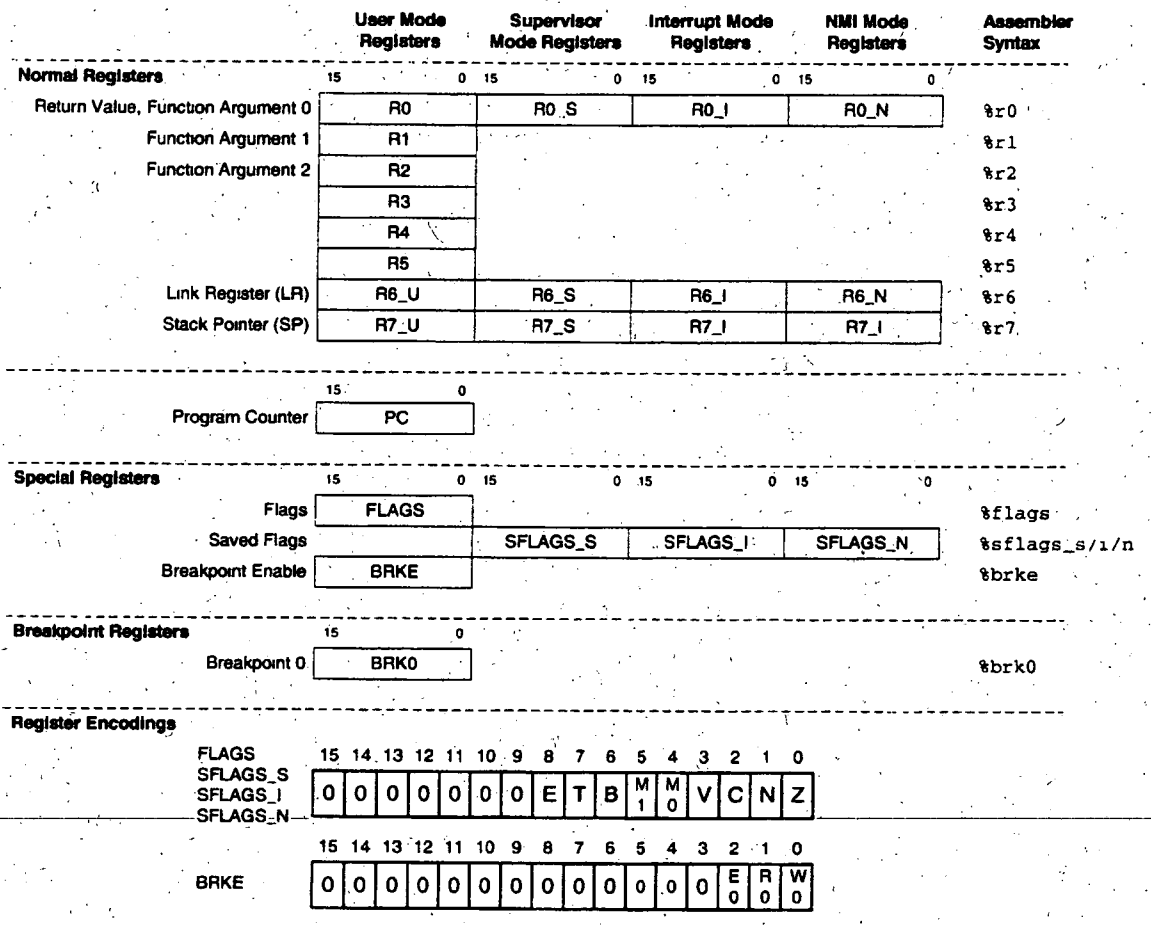
Here is a block diagram for XAP4a, which is a minimum implementation of the XAP4 instruction set, using a single physical bus for all memory accesses (Prog and Data) :




XAP4a has a 2-stage pipeline.
It supports a single memory bus (16bit data (in & out), 16bit address)
Instructions are implemented in the minimum number of clocks.
RRR & RRI instructions take 1 clock (16-bit instr) or 2 clocks (32-bit instr)
LD, ST instructions take 2 clocks (16-bit instr), or 3 clocks (32-bit instr).

There is more information on how a single ASIC can support multiple on-chip processors in the SIF section at the end of this specification.

# 5 Programmer's Model



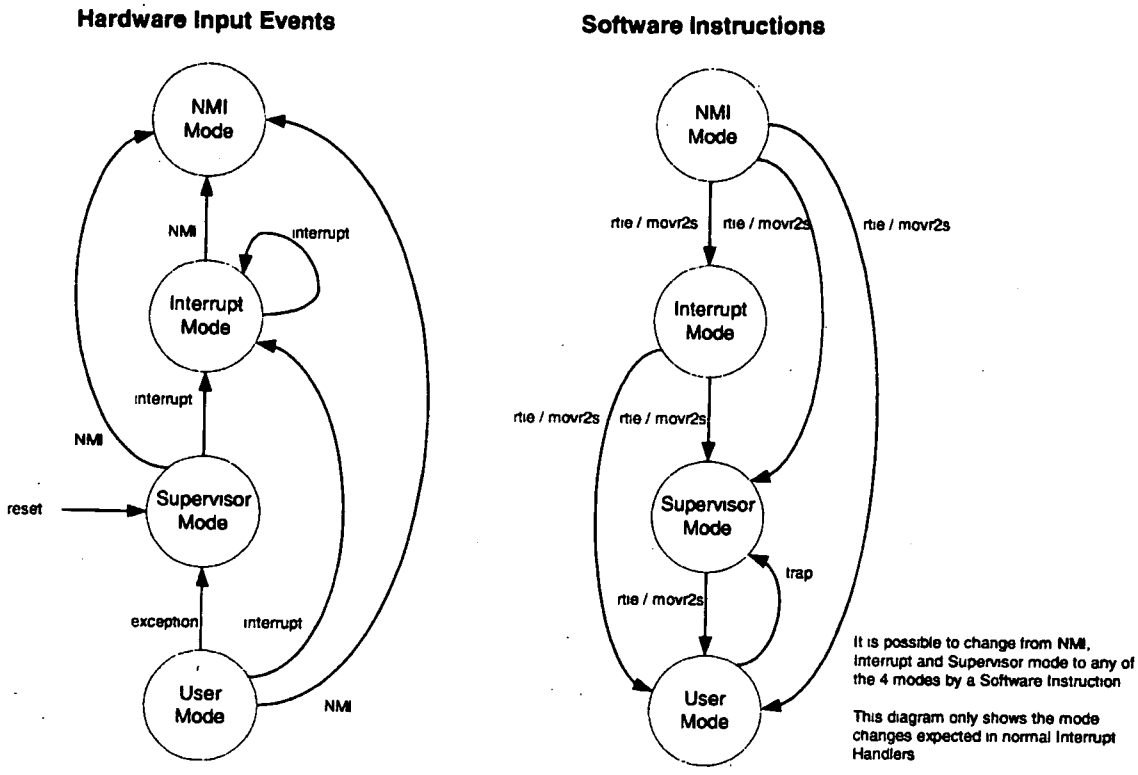

## 5.1 Processor Mode

There are 4 modes of operation:

- User
- Supervisor
- Interrupt
- NMI (Non-Maskable Interrupt)

User mode allows unknown software to be run safely without affecting the operation of the interrupt mode code. The other 3 modes are privileged modes.

The diagram below shows how the mode changes will take place.




We expect the 4 modes to be used as follows:

- NMI Mode – Special Interrupt mode for supporting non-maskable interrupts.
- Interrupt Mode – Hardware Drivers (IVC hardware generates defined interrupt addresses for specified events)
- Supervisor Mode – Operating System
- User Mode – Application Software

The movr2s instruction can be used to change from any privileged mode to any mode.

It is expected that the majority of code will execute in User Mode.

The processor resets to Supervisor mode. The initialisation code is then responsible for initialising the stack and starting User Mode. The mode switch is implemented using the `movr2s` instruction to write to the FLAGS register.

User mode can only access User registers. The Supervisor, Interrupt and NMI modes are known as privileged modes. They can access their own and the User Mode registers. Supervisor Mode cannot access the Interrupt or NMI Mode registers, and vice-versa. In order to initialise Interrupt and NMI mode registers, programmers must disable interrupts and manually switch to Interrupt mode by writing to the FLAGS register.

Code executed in User Mode cannot directly write to the FLAGS register.

## 5.2 Processor State

The processor can be in two states:

- Sleeping
- Awake

When the processor is awake, SIF accesses can only take place when the CPU is executing a sif instruction.

There are two sleep modes. One that allows SIF accesses and one that doesn't.

## 5.3 Run State

The state of the program refers to whether the core is executing instructions or not. This is deteremined by RUN_STATE[1:0] in the XAP4 Status Register (see 9.6.3). This is decoded as the following 4 states:

0 Run Continuous

1 Run to Breakpoint

2 Stopped

3 Single Step

RunContinuous is the normal state used for program execution. The other 3 states will only be encountered during interactive debugging.

## 5.4 Normal Registers

Registers R0, R6 and R7 are shadowed such that there is one of each of these registers per processor mode. For these registers, the actual register accessed by an instruction depends on the mode in which the processor is operating.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| `%r0` | 000 | R0_U or<br>R0_S or<br>R0_I or<br>R0_N | Dependent on processor mode.<br>Used as :<br>• return value<br>• function argument 0 |
| `%r1` | 001 | R1 | Used as :<br>• function argument 1 |
| `%r2` | 010 | R2 | Used as :<br>• function argument 2 |
| `%r3` | 011 | R3 | |

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| %r4 | 100 | R4 | |
| %r5 | 101 | R5 | |
| %r6 | 110 | R6_U or R6_S or R6_I or R6_N | Dependent on processor mode. Used as : • Link Register (LR) |
| %r7 | 111 | R7_U or R7_S or R7_I | Dependent on processor mode. Used as : • Stack Pointer (SP) |

Any pair of adjacent registers can be joined to form a 32-bit accumulator. These are used by the mult, div, rem, divrem, shift and rotate instructions. Such accumulators are referred to by the lower register of the pair. e.g :

- {R2, R1} is referred to as %r1
- {R5, R4} is referred to as %r4

The C compiler uses R7 as Stack Pointer and R6 as Link Register. R0, R6 and R7 are shadowed in Supervisor, Interrupt and NMI modes.

At any one time, a register is only used for a single variable (whether it is 8 or 16-bit). All arithmetic and logical operations operate on full 16-bit words. Load, Store and Compare instructions have 16-bit and 8-bit forms (zero extended), allowing programs to operate on 8-bit variables.

## 5.5 Breakpoint Registers

Software development on XAP4 can implement break points 2 ways :

- Swap normal instruction with brk instruction in memory. This cannot support conditional breaks. This can only be used in memory that supports individual word writes. 'xIDE for XAP4' chooses to use these where possible, as they are an unlimited resource.
- Use the Break register. This can support conditional breaks. This can be used with any kind of memory (because the memory itself is not modified).

The XAP4 contains one breakpoint register, BRK0, that can be configured to stop the processor when an address is read, written, or executed.

Providing privileged modes access to the breakpoint register allows Interrupt Mode debuggers to be implemented that allow User Mode tasks to be debugged, without needing to stop the processor. This allows, for example, the main operating system to remain running when debugging User Mode applications.

The bits in the BRKE register are described below.

| *Bit(s)* | *Flag(s)* | *Name* | *Description* |
|---|---|---|---|
| 0 | W0 | Write | If set, an instruction making a memory write to the address matching the corresponding BRKn register will halt the processor after the instruction has executed. The PC will point at the next instruction to be executed. |
| 1 | R0 | Read | Similar to above, but for memory reads. |
| 2 | E0 | Execute | If set, an instruction fetch from the address matching the corresponding BRKn register will halt the processor before the instruction is executed. The PC remains at the address of the instruction that caused the break. |

Note: the BRKE register is one of the Special Registers. The special registers are described in section 5.6.

The above break conditions will only halt the processor if the following conditions are met :

- (B flag is 0) or (Processor is in a privileged mode (Supervisor, Interrupt, NMI))
- RUN_STATE = RunToBreak

If the B flag is 1 and the processor is in User mode, it will not halt, but will generate a Break exception instead (see section 6.7.5).

The breakpoint registers can be written and read by privileged modes using the `movr2b` and `movb2r` instructions, see below. The `movs2r` and `movr2s` instructions allow access to the BRKE register.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| `%brk0` | 000 | BRK0 | |
| | 001 - 111 | *reserved* | |

## 5.6 Special Registers

The `movs2r` and `movr2s` instructions allow Interrupt, Supervisor and NMI mode access to the Special Registers. These are shown in the table below.

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| `%flags` | 000 | FLAGS | |
| `%sflags_s` | 001 | SFLAGS_S | Saved Flags. The processor copies FLAGS into this register when an |

| *Assembler Syntax* | *Instruction Encoding* | *Register Name* | *Notes* |
|---|---|---|---|
| | | | exception happens. |
| `%sflags_i` | 010 | SFLAGS_I | Saved Flags. The processor copies FLAGS into this register when an interrupt happens. |
| `%sflags_n` | 011 | SFLAGS_N | Saved Flags. The processor copies FLAGS into this register when an NMI happens. |
| `%brke` | 100 | BRKE | Break Enable. |
| `%r0_u` | 101 | R0_U | R0 User Mode Register |
| `%r6_u` | 110 | R6_U | R6 User Mode Register |
| `%r7_u` | 111 | R7_U | R7 User Mode Register |

The format of the Flags registers is descibed in section 5.7.

## 5.7 Flags

The table below describes the layout of the FLAGS, SFLAGS_S, SFLAGS_I and SFLAGS_N registers.

| *Bit* | *Flag* | *Name* | *Description* |
|---|---|---|---|
| 0 | Z | Zero | Set if result is zero (Rd = 0) |
| 1 | N | Negative | Set if result is negative (Rd[15] = 1) |
| 2 | C | Carry | Set if unsigned arithmetic overflowed |
| 3 | V | Overflow | Set if signed arithmetic overflowed |
| 5:4 | M[1:0] | Mode | 00: Supervisor Mode<br>01: User Mode<br>10: Interrupt Mode<br>11: NMI Mode |
| 6 | B | Break | If set and the processor is in User Mode, a break as a result of a brk instruction or the BRK registers will throw the Break exception. |
| 7 | T | Single Step | If set, each User Mode instruction that is executed will throw the SingleStep exception. |
| 8 | E | Interrupt Enable | If set, normal interrupts are enabled. (NMI cannot be disabled) |

When in Supervisor, Interrupt or NMI modes, the entire FLAGS register can be modified by executing the movr2s instruction. The movs2r instruction allows the FLAGS register to be read. The `irqe` and `irqd` instructions can be executed in Supervisor, Interrupt or NMI Mode. They are used to enable and disable interrupts.

The mode bits of the FLAGS register are automatically changed when interrupts or exceptions happen. The other bits remain unchanged. At reset the processor will start executing code in Supervisor mode.

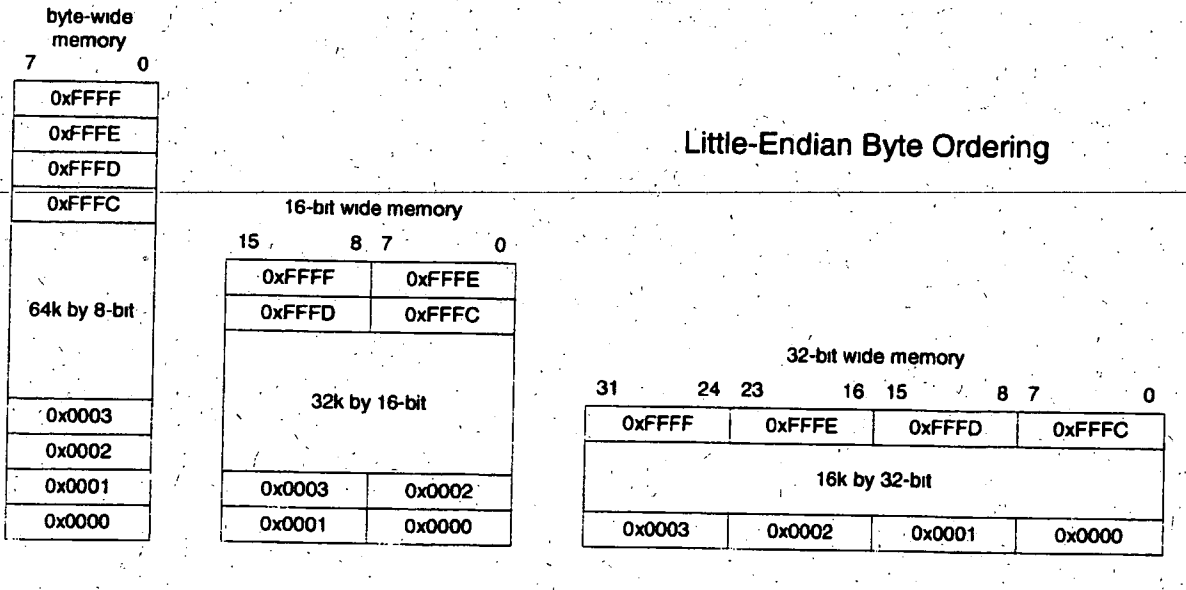
The condition flags, Z, N, C, and V will be modified as instructions are executed. See the instructions description for details of the instructions that modify flags.

The carry versions of add, subtract and compare instructions update the Z flag as follows :

- If (result=0) and (Z=1), set Z=1. Otherwise set Z=0.
- This means that if (Z=0) before the instruction, then Z will always be 0 after the instruction (regardless of the instruction result).
- This is useful for 32-bit arithmetic.

## 5.8 Memory Model

The memory model has a 16-bit address space (64 kBytes), containing both code and data. The address space is byte addressed. All pointers are kept as full 16-bit byte addresses, and will thus fit in a 16-bit register.




Little-Endian Byte Ordering

All instructions are aligned to a 16-bit boundary (i.e. bit 0 of the address is 0), but it is not necessary for 32-bit instructions to be aligned to a 32-bit boundary.

All data objects are unaligned. This means that 8, 16 and 32-bit data can have any byte address. The diagram above only shows aligned data objects.

All words use Little Endian byte ordering, i.e. the low bits of a 16-bit or 32-bit word are at the low byte address.

## 5.9 Reset State

All registers and flip-flops in XAP4 are asynchronously set or reset when the RESETB input pin goes low. Wherever possible, the registers and flip-flops should be reset and not set.

Memories (internal and external) cannot be reset and so will not be.

At reset, the XAP4 will be in the following state :

- Program Counter = 0 (i.e. code execution starts from address 0)
- Interrupts disabled
- In Supervisor mode
- Program Running
- Processor Awake
- SIF in Normal mode

## 5.10 Calling Convention

The bsr.p or bsr.a instruction is used to make function calls. The effect of bsr.p or bsr.a is to transfer the address of the next instruction to the link register, R6, and to transfer the destination address to the PC. Control is returned to the calling function when the contents of R6 is transferred to PC. In practice, the compiler generates the bsr.p instruction.

The calling convention for the XAP4 is :

- The called function does not need to preserve the contents of R0, R1, R2, and R6. R6 = Link Register.
- The called function is responsible for preserving the remaining registers (R3-R5, R7) across function calls. This convention is implemented in the xap4-gcc C compiler. It is not a hardware property.
- R7 (Stack Pointer) is used to operate a fully covered stack, so interrupt handlers don't need to allow for a partially covered stack as in XAP2 systems. A fully covered stack is when function data is always at positive or zero offset from the stack pointer. A partially covered stack allows negative offsets to be used.
- Arguments are passed to functions in R0, R1, R2 then on the stack. When an 8-bit value is passed to a function in a register, the whole register is used. The upper bits are sign or zero extended according to the type. Structures that are passed in registers are converted to a sequence of 16-bit words.
- The called function places the C function return value in R0 (convention implemented in xap4-gcc).
- Special functions may store extra return data in R1 (e.g long values).
- Code is compiled independent of processor mode.

- When the processor is interrupted, the contents of R1-R5 must be preserved. This is not needed for (R0, R6, R7) as they have hardware shadow registers.
- C functions must not corrupt (R3-R5, R7), but may corrupt (R0, R1, R2, R6).

## 5.11 Application Models

The XAP4 can support various application models.

### *5.11.1 Statically Linked*

This is the basic application model where the locations of all functions and data are known at link time.

### *5.11.2 Dynamically Loaded*

The XAP4 supports systems where applications can be loaded to memory locations that are not known at link time. Statically allocated variables are accessed relative to the PC register.

The relative positions of code and data can be varied at load-time. PC relative addressing for code and constants (probably stored in Flash) means only minimal load time fixups are needed.

If the same program is being run in more than one instance, then the code/constants only need to be loaded once.

Absolute addressing can be used to access code and data at fixed addresses. This means that no load time fixups are required to access absolute code and data locations within the system.

Fixups will be necessary when static initialisers contain an address of a static variable. The location of the static variable is not known until link time, hence the loader will need to fixup the initialisation constant. This code would require a load time fixup when used as part of a dynamically loaded application.

```
static int stVar;
static int * stPtr = &stVar; // Fixup needed because of &
```

# 6 Interrupts and Exceptions

This section describes the interrupt and exception handling mechanisms of XAP4. The objectives of the design are:

- to keep the XAP4 core as simple as possible, in keeping with the general design philosophy, and to minimise the burden of implementing simple controller applications.
- to provide a level of capability that will support a wide variety of operating systems, from simple round-robin schedulers, to modern multi-tasking operating systems such as Micrium uC/OS-II.
- to support debuggers that allow users to interactively debug a User Mode application without halting an underlying operating system.

In the description below the following terms are used.

- **Handler** – a piece of code that handles the response to an interrupt or exception.
- **Exception** – a special event triggered by some action of the XAP4 core, e.g., dividing by zero, or undefined instruction.
- **Interrupt** – a special event triggered by some action external to the XAP4 core, e.g., a timer reaching its terminal count, or a serial port receiving a character.
- **Vector** – the mapping between an interrupt or exception identifier and the corresponding handler.

## 6.1 Overview

The XAP4 core has four operating modes:

- User Mode
- Supervisor Mode
- Interrupt Mode
- NMI Mode

Exceptions may be thrown by code executing in User Mode and cause a switch to Supervisor Mode. Exceptions cause interrupts to be disabled. The exception handler can re-enable them using the `irqe` (Interrupt Enable) instruction.

The XAP4 has two types of interrupt:

- Interrupts – can be enabled and disabled
- Non-Maskable Interrupts – cannot be disabled

Interrupts cause the processor to switch to Interrupt Mode and to disable further interrupts. Interrupts can be enabled and disabled using the `irqe` (Interrupt Enable) and `irqd` (Interrupt Disable) instructions.

Non-Maskable Interrupts force a switch to a special NMI Mode. Just enough information is preserved in the processor to allow return to the interrupted code. Since NMI Interrupts cannot be disabled, special care should be taken in writing NMI handlers and in the design of the hardware to ensure that the software can handle multiple NMIs (or simply disallow multiple NMIs).

Note that as NMI Mode is a special form of Interrupt Mode it also shares a single stack with Interrupt Mode (SP_I). Systems which implement NMI handlers should allow space on the Interrupt Mode stack for any NMI handlers that may be called during execution of Interrupt handlers.

Supervisor Mode is never re-entrant. Interrupt Mode can be re-entrant if interrupts are re-enabled within the Interrupt Mode handler.

## 6.2 Exception Processing

Exceptions in the XAP4 are precise exceptions, in that we can know the exact instruction that caused the exception. This allows the operating system to be very precise in its handling of the exception.

The cost of precise exceptions comes when implementing a pipelined processor core, since there can be several instructions in flight at the same time, and at each stage in the pipeline exceptions can be triggered. This requires extra complexity in the processor core to track the address of each instruction as it progresses through the pipeline.

This approach behaves the same if implemented as a single-scalar machine, or as a super-scalar machine.

On entry to an exception:

- The processor will be in Supervisor Mode.
- LR points to the next instruction to be executed when the Handler finishes.
- For the Trap exception, R0 will be set to the argument of the `trap` instruction.
- For all other exceptions, the address of the instruction that caused the exception is placed in R0.
- The contents of the FLAGS register just before the exception will be copied into SFLAGS_S.
- Interrupts are disabled.

For most exceptions on entry to the handler R0 will point to the instruction that caused the exception. This is necessary because instructions are variable length and it is a non-trivial task to determine the start of the instruction that caused the exception without

walking back through the instruction stream. In some circumstances this could cause other exceptions as well (e.g., if the instruction that caused the exception was the first instruction in a code segment then accessing the previous word could cause a memory protection exception).

Exceptions can only happen in User Mode and cause a switch to Supervisor Mode. In the Supervisor, NMI and Interrupt Modes exceptions have no effect (the `trap` instruction behaves as a `nop`).

The complete list of exceptions is given in section 6.7.

The core processes exceptions in the following manner:

```
if exception thrown && FLAGS.Mode == UserMode) then
      SFLAGS_S = FLAGS
      LR_S     = PC
      if (ExceptionNum is TRAP) then
            R0_S   = instruction argument
      else
            R0_S   = LastPC
      endif
      FLAGS.Mode  = SupervisorMode
      FLAGS.E     = 0
      PC          = (ExceptionNum * 4) + IVTB
endif
```

Returning from an exception is done by the `rtie` instruction (see section 6.4 below).

The null exception handler is simply the `rtie` instruction alone, which can be accommodated within the Interrupt Vector Table (see section 6.6 below).

## 6.3 Interrupt Processing

The processor has a single IRQ input and a single NMI input. If interrupts are enabled, activating the IRQ input will cause the processor to switch to Interrupt Mode when the current instruction finishes and query the interrupt controller for an interrupt number, IRQ_NUM. This number is converted in to a word offset, and is then used to index (plus offset 0x0040) into the IVT to point to the interrupt handler.

Non-Maskable Interrupts can occur at any time and cannot be disabled. The additional shadow registers in NMI mode ensure that an NMI handler can restore the processor to its state immediately prior to the NMI event.

The interrupt vector controller is responsible for implementing any priority based schemes.

The number of interrupt vectors is limited by the width of the irq_num bus. The irq_num bus is 4-bit and so supports up to 16 different interrupt vectors.

On entry to an interrupt:

- The processor will be in Interrupt (or NMI) Mode.
- LR points to the next instruction to be executed when the handler finishes.
- The contents of the FLAGS register just before the interrupt will be copied into SFLAGS_I (or SFLAGS_N for NMI mode).
- The E flag is cleared, which disables further Interrupts. In NMI mode further Interrupts are effectively disabled because they are ignored.

The core processes interrupts in the following manner:

```
if (posedge NMI) then
      SFLAGS_N     = FLAGS
      FLAGS.Mode   = NMIMode
      LR_N         = PC
      PC           = 0x0004 + IVTB
elseif (FLAGS.E && IRQ) then
      SFLAGS_I     = FLAGS
      FLAGS.E      = 0
      FLAGS.Mode   = InterruptMode
      LR_I         = PC
      PC           = 0x40 + (IRQ_NUM * 4) + IVTB
endif
```

## 6.4 Returning from Interrupts and Exceptions

The `rtie` instruction is used to return from interrupts and exceptions and is implemented as:

```
if (FLAGS.Mode == NMIMode) then
      FLAGS = SFLAGS_N
      PC    = LR_N
elseif (FLAGS.Mode == InterruptMode) then
      FLAGS = SFLAGS_I
      PC    = LR_I
elseif (FLAGS.Mode == SupervisorMode) then
      FLAGS = SFLAGS_S
      PC    = LR_S
endif
```

Thus the null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code.

## 6.5 Nested Interrupts

To support nestable interrupts some additional work is required of the handler to save, and later restore, the processor state.

On entry:

```
push        {%r0-%r6}, #2      // Save registers R0-R6 and
                               // allocate 1 word on the stack.
movs2r      %r0, %sflags_i     // Copy SFLAGS_I to R0.
st.i        %r0, @(0,%r7)      // Store R0 on stack.
irqe                           // Re-enable interrupts.
```

Then on exit we restore the original registers and flags:

```
irqd                           // Disable interrupts
ld.i        %r0, @(0,%r7)      // Get SFLAGS_I from stack
movr2s      %sflags_i, %r0     // Restore SFLAGS_I
pop         {%r0-%r6}, #2      // Restore registers R0-R6
                               //  & free stack space
rtie                           // Return from interrupt
```

Note that the exact register lists for the push and pop can be tailored for specific handlers to minimise stack usage and interrupt response time.

NMI handlers that are expected to return control to the interrupted process may use similar entry and exit code, with the omission of the `irqe` and `irqd` instructions.

## 6.6 Interrupt Vector Table

The Interrupt Vector Table (IVT) contains instructions. These will often be bra.p to the relevant handler routines. The IVT manages both exceptions and interrupts. The IVT sits in memory with its base at address 0.

This means that reset will set the PC register to 0.

Each entry in the table is one 32-bit word, and contains executable code. This is typically a branch to the start of the actual handler, but may simply be the null handler (the `rtie` instruction).

The layout of the IVT is shown below. The unused exceptions are for future expansion.

| *Address* | *Processor Mode* | *Vector* | |
|---|---|---|---|
| 0x0000 | S | 0 | Reset |
| 0x0004 | N | 1 | Non-Maskable Interrupt |
| 0x0008 | S | 2 | Trap Exception |

| *Address* | *Processor Mode* | *Vector* | |
|---|---|---|---|
| 0x000C | S | 3 | SingleStep Exception |
| 0x0010 | S | 4 | Break Exception |
| 0x0014 | S | 5 | UnknownInstruction Exception |
| 0x0018 | S | 6 | *reserved* Exception |
| 0x001C | S | 7 | *reserved* Exception |
| 0x0020 | S | 8 | DivideByZero Exception |
| 0x0024 | S | 9 | NullPointer Exception |
| 0x0028 | S | 10 | IllegalInstruction Exception |
| 0x002C | S | 11 | IllegalInstructionAccess Exception |
| 0x0030 | S | 12 | IllegalDataAccess Exception |
| 0x0034 | S | 13 | IllegalCodeAddress Exception |
| 0x0038 | S | 14 | *reserved* Exception |
| 0x003C | S | 15 | *reserved* Exception |
| 0x0040 | I | 0 | Int00 Interrupt |
| 0x0044 | I | 1 | Int01 Interrupt |
| 0x0048 | I | 2 | Int02 Interrupt |
| ... | ... | ... | ... |

## 6.7 Exceptions

Further details of each exception are described below. It is expected that the OS will kill (and maybe restart) applications throwing exceptions 8 (DivideByZero) onwards.

### *6.7.1 Reset*

While not technically an exception, placing the reset vector in the exception group leads to a simpler processor model.

### *6.7.2 NMI*

The NMI handler is called when an edge is detected on the NMI input. This would be used for system-critical events that cannot be ignored (unlike ordinary interrupts that can be disabled) such as loss of power, fatal hardware failure, etc.

### *6.7.3 Trap*

Software trap instructions may be used to call an operating system. All `trap` instructions take an argument, either a register or an immediate. On entry to the trap handler, LR points to the next instruction, and R0 is set to the argument of the instruction (e.g., OS function number).

The following instructions cause this exception

- `trap.r`

- trap.i

### 6.7.4 SingleStep

When the T bit is set in the FLAGS register, this exception is triggered after every User Mode instruction, and only if that instruction has not caused any other exception (e.g., DivideByZero). The LR points to the next instruction to be executed and R0 points to the instruction that caused the exception. This supports remote debugging and single stepping of User Mode applications.

### 6.7.5 Break

The Break exception can be caused by either a brk instruction or by a break condition being satisfied in the BRK registers.

Break events are handled according to this logic.

```
if FLAGS.Mode == User && FLAGS.B == 1 then
      throw break exception
else if RUN_STATE = RunToBreak then
      halt
else
      nop
endif
```

RUN_STATE cannot be set to RunToBreak unless the XAP4 is in Debug mode. Debug mode is enabled with the SIF. The B flag takes precedence over the debug mode action.

It is intended that the xIDE debugger for the hardware emulator will always :

- Leave XAP4 in Debug mode
- Start the XAP4 with RunToBreak (not with Run continuously)

Any break conditions will halt the processor unless the processor is in User Mode and the B flag is set.

The brk instructions are not protected instructions. An operating system can prevent a User Mode task from throwing a Break exception by clearing the B bit of the FLAGS register. For normal debugging of User Mode code the T and B flags are not set and the Simulator and Emulator will use instruction replacement and the BRK registers to implement breakpoints.

### 6.7.6 UnknownInstruction

All opcodes that do not correspond to a valid instruction trigger this exception.

LR points to the next instruction to be executed and R0 points to the undefined instruction just executed.

This exception has two useful functions. Firstly, it can quickly trap bad code if it causes the processor to start executing data or other non-code.

Secondly, and perhaps more importantly, it is a simple, proven, mechanism for supporting pseudo-instructions in a processor. This has two significant benefits:

- It allows any version of the XAP4 core to support, through software emulation, any custom instruction of any other version of the XAP4 core, albeit at a slower rate of execution. Note this includes dynamically updating a system with new instructions simply by adding more functionality to the instruction emulation code.
- It supports experimentation and analysis of any new instruction simply by adding emulation code, perhaps to evaluate code size or feature exploration.

The immediate forms of all branches should only have even addresses or offsets. If the assembler generates invalid code with odd addresses or offsets, the processor will generate an UnknownInstruction exception.

### 6.7.7 DivideByZero

All forms of divide and remainder instructions check the value of the denominator. If it is zero, then this triggers a DivideByZero exception. LR points to the next instruction to be executed and R0 points to the divide instruction just executed.

The following instructions can cause this exception:

- `div.*`
- `rem.*`
- `divrem.*`

### 6.7.8 NullPointer

The XAP4 debug module generates a NullPointer exception if a data access to address 0 is attempted. Address 0 contains the reset vector and therefore cannot contain data.

After a NullPointer exception, LR points to the next instruction to be executed and R0 points to the instruction just executed.

The following instructions can cause this exception:

- `ld.i, ld.8z.i`
- `ld.p, ld.8z.p`
- `ld.r, ld.8z.r`
- `st.i, st.8.i, stz.i, stz.8.i`
- `st.p, st.8.p, stz.p, stz.8.p`
- `st.r, st.8.r, stz.r, stz.8.r`

- `swap.i`
- `blkcp.r, blkcps.r`
- `blkst.r, blkst.8.r`
- `push`
- `pop, pop.ret`

### 6.7.9 IllegalInstruction

The XAP4 core will execute most code in User Mode. This mode is restricted in its access rights to certain instructions; executing any protected instruction in User Mode will trigger this exception. LR points to the next instruction to be executed and R0 points to the instruction that caused the exception.

The following instructions cause this exception, because they provide access to the FLAGS register and could be used to create mode changes:

- `rtie`
- `movr2s, movs2r`
- `movr2b, movb2r`
- `irqe, irqd`

The following instructions cause this exception, because they alter the behaviour CPU core.

- `sleepsif, sleepnop`
- `halt`
- `sif`

The `sif` instruction is not allowed in User Mode, because it can alter the timing of accesses from external devices to the registers and memory of the processor. Systems wishing to allow User Mode to control when SIF accesses can take place, would need to provide a Supervisor Mode routine accessible via the trap instruction.

### 6.7.10 IllegalInstructionAccess and IllegalDataAccess

The MODE[1:0] signal from xap4_core to xap4_mmu indicates whether the xap4_core is in User, Supervisor, Interrupt or NMI Mode. The xap4_core has a 'memory exception' input that allows the xap4_mmu to indicate when a memory access is attempted that is illegal in the current operating mode.

These exceptions are triggered by an external Memory Management Unit (XAP4_mmu) in response to the XAP4 core accessing memory that violates the current access rights (e.g., reading from memory that does not belong to the currently executing process, or writing to memory that is tagged as read-only).

IllegalInstructionAccess is thrown when an instruction fetch violates the access rights. IllegalDataAccess is thrown when an instructions data read or write violates the access rights.

On entry to the handler LR points to the next instruction to be executed and R0 points to the instruction that caused the exception. It is intended that the XAP4_mmu places the address that caused the exception in some memory mapped register accessible to the exception handler code.

These exceptions are important in virtual memory systems (implemented in software with an appropriate hardware MMU). When a program accesses data (or code) in its memory space that currently belongs to another task, this exception will be triggered since it violates the memory protection afforded to the other task. The virtual memory manager can then swap out the sleeping task's memory chunk, in the active task's memory chunk, and then restart the current instruction.

IllegalDataAccess can be caused by the following instructions:

- `ld.i, ld.8z.i`
- `ld.p, ld.8z.p`
- `ld.r, ld.8z.r`
- `st.i, st.8.i, stz.i, stz.8.i`
- `st.p, st.8.p, stz.p, stz.8.p`
- `st.r, st.8.r, stz.r, stz.8.r`
- `swap.i`
- `blkcp.r, blkcps.r`
- `blkst.r, blkst.8.r`
- `push`
- `pop, pop.ret`

### *6.7.11 IllegalCodeAddress*

All instructions are half-word aligned, such that PC[0] = 0. This exception is triggered when an odd value is written to PC, (i.e., PC[0] = 1). LR points to the next instruction to be executed (i.e., the invalid address) and R0 points to the branch instruction that caused the exception.

The following instructions can cause this exception:

- `bra.r`
- `bsr.r`
- `pop.ret`
- `rtie`

Note: it is not possible for the immediate forms of the branch instructions to throw this exception, because they cannot encode illegal addresses.

# 7 Addressing Modes

ALU instructions have 2 addressing modes:

- Register, Register, Immediate (RRI)
- Register, Register, Register (RRR)

In each case the result is put into the Rd register. The registers are always 16-bit.

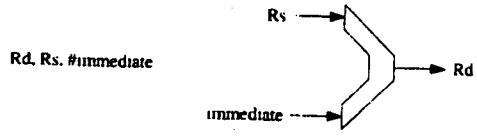
The only instructions that can read memory are ld.*, pop, pop.ret, swap.i, blkcp*.

The only instructions that can write to memory are st.*, stz.*, push, swap.i, blkcp*, blkst*.

Hence the XAP4 has a load-store architecture.

## 7.1 Register, Register, Immediate

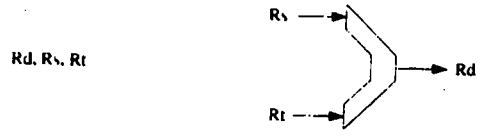
One input to the ALU comes from the Rs register and one from an immediate value that is part of the instruction word. The input and output register can be specified separately.




In 32-bit instructions, the immediate is 16-bit. In 16-bit instructions, the immediate can be a variety of lengths (but is normally 8-bit).

## 7.2 Register, Register, Register

Both input registers (Rs, Rt) and the output can be specified separately.



## 7.3 Load, Store

There are 3 forms of the basic load and store instructions.

### 7.3.1 Displacement Addressing

Displacement Addressing is implemented by the '.i' form of the load and store instructions. The byte address is calculated as:

offset + Ra

### 7.3.2 PC-Relative Addressing

Displacement Addressing is implemented by the '.p' form of the load and store instructions. The byte address is calculated as:

offset + PC

### 7.3.3 Indexed Addressing

Indexed Addressing is implemented by the '.r' form of the load and store instructions. The byte address is calculated as:

(Rx * n) + Ra

| | |
|---|---|
| 8-bit data | n = 1 |
| 16-bit data | n = 2 |

# 8 Instruction Set

## 8.1 Notation Conventions

In the following sections the following conventions are used to describe various aspects of the instruction set.

- Registers are prefixed by %.
- Rd is the destination register of the instruction
- Rs, and Rt are the primary and secondary registers containing source operands
- Ra, and Rx are registers containing a base address and index for an operand. An offset may also be added to the register contents to form the actual address of the operand. Ra is a byte address for the base. Rx is a scaled address for the index (*2 for 16-bit, *1 for 8-bit data). Both Ra and Rx are unsigned numbers.
- Immediate[] indicates a constant operand encoded as part of the instruction. In the documentation #immediate[highbit:lowbit]s or #immediate[highbit:lowbit]u is used to distinguish between signed and unsigned immediates. Immediates are prefixed by #. Note, the # is not used for address offsets (e.g @(0, %r1))
- Address[] indicates an absolute address. This is a byte-address, unless specified otherwise. Addresses are prefixed by @. Absolute addresses are always interpreted as 16-bit unsigned numbers.
- Base addresses stored in Ra should be interpreted as byte addresses. Index offsets stored in Rx should be multiplied by (1, 2) respectively for (8, 16) bit data to get the desired byte address offset.
- Offset[] indicates an address offset. This may be relative to the program counter (in branch instructions, for example), or to a pointer read from a register (the stack pointer, for example). Offsets can be signed or unsigned numbers (indicated by appended s or u). Offsets are byte-address values, unless specified otherwise. In the documentation offset[highbit:lowbit]s or offset[highbit:lowbit]u is used to distinguish between signed, unsigned and index offsets. A signed offset should be sign extended to a 16-bit word before calculating an address. An unsigned offset should be zero extended to a 16-bit word before calculating an address.

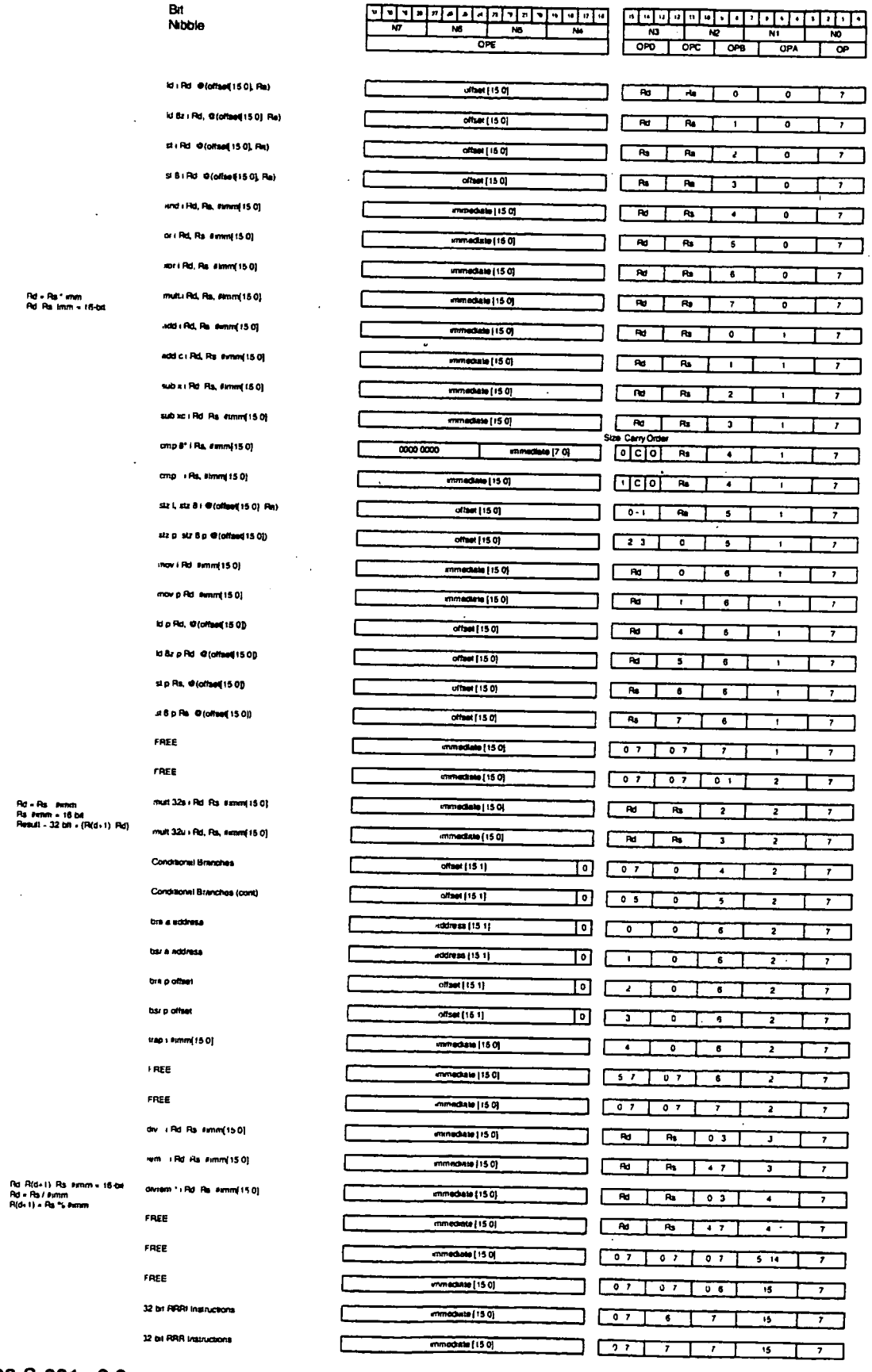
## 8.2 Structure of 16-bit Instructions

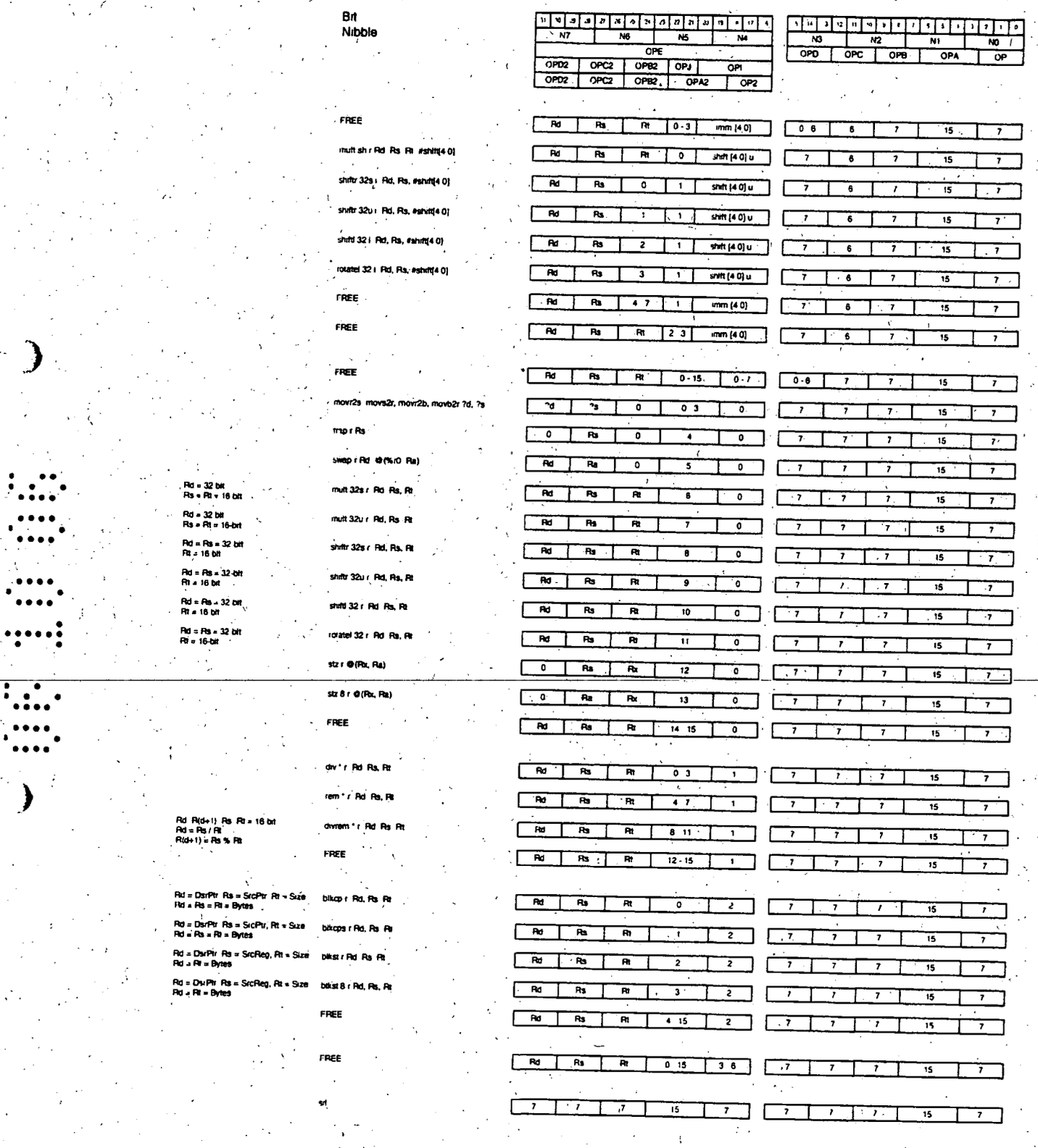
Bit
Nibble

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N3 | | | | N2 | | | | N1 | | | | N0 | | | |
| OPD | | | | OPC | | | | OPB | | | | OPA | | | OP |
| OPD | | | | OPG | | | | OPF | | | | | | | OP |
| OPD | | | | OPG | | | | OPH | | | | OPA | | | OP |

| | | | | |
|---|---|---|---|---|
| ld.i Rd, @(offset[8.1], Ra) | Rd = R0, R1, R2, R3 | Ra | Rd | offset [8.1] u | 0 |
| ld.i Rd, @(offset[8.1], Ra) | Rd = R4, R5, R6 | Ra | 0 2 | offset [8.1] u | 1 |
| ld.8z.i Rd, @(offset[7.0], Ra) | Rd = R0 | Ra | 3 | offset [7.0] u | 1 |
| st.i Rs, @(offset[8.1], Ra) | Rs = R0, R1, R2, R3 | Ra | Rs | offset [8.1] u | 2 |
| st.i Rs, @(offset[8.1], Ra) | Rs = R4, R5, R6 | Ra | 0 2 | offset [8.1] u | 3 |
| st.8.i Rs, @(offset[7.0], Ra) | Rs = R0 | Ra | 3 | offset [7.0] u | 3 |
| mov.i Rd, #imm[7.0] | | Rd | 0 | immediate [7.0] u | 4 |
| cmp.i Rs, #imm[7.0] | | Rs | 1 | immediate [7.0] u | 4 |
| and.i Rd, Rd, #imm[7.0] | | Rd / Rs | 2 | immediate [7.0] u | 4 |
| add.i Rd, Rd #imm[7.0] | | Rd / Rs | 3 | immediate [7.0] s | 4 |
| beq, bne, ble.s, bgt.s #imm[8.1] | | 0 | 0 3 | offset [8.1] s | 5 |
| blt.s, bge.s, ble.u, bgt.u #imm[8.1] | | 1 | 0 3 | offset [8.1] s | 5 |
| FREE | | 2 | 0 3 | offset [8.1] s | 5 |
| bra.p #imm[8.1] | | 3 | 0 | offset [8.1] s | 5 |
| push, pop, pop ret - {regist[3.0]}, #offset[4.1] | | 3 | 1 3 | regist [3.0] | offset [4.1] u | 5 |
| shiftr.s.i, shiftr.u.i, shiftl.i, rotatel.i Rd Rs, #shift | | 4 7 | Rs | Rd | shift [3.0] u | 5 |
| or.i, stz.i, mov.i add.i | | 4 7 | 7 | Rd | imm [ ] u | 5 |
| cmp.8.i, cmp.i, add.i | | 4 7 | Rd / Rs | 7 | imm [3.0] u | 5 |
| ld.r, ld.8z.r Rd, @(Rx Ra) | Note - no ld.8s.r | Rd | Ra | Rx | 0 1 | 6 |
| st.r, st.8.r - Rs, @(Rx, Ra) | | Rs | Ra | Rx | 2 - 3 | 6 |
| and.r or.r, xor.r, mult.r - Rd, Rs, Rt | mult.r Rd = Rs * Rt<br>Rd = Rs * Rt >> 16 bit | Rd | Rs | Rt | 4 - 7 | 6 |
| add.r, add.c.r, sub.r sub.c.r Rd, Rs, Rt | | Rd | Rs | Rt | 8 11 | 6 |
| shiftr.s.r, shiftr.u.r, shiftl.r rotatel.r Rd, Rs, Rt | | Rd | Rs | Rt | 12 15 | 6 |
| cmp.8.r, cmp.8c.r cmp.r, cmp.c.r - Rs, Rt | | 7 | Rs | Rt | 4 7 | 6 |
| bra.r/ bsr.r Ra | | 7 | Ra | 0 - 1 | 12 | 6 |
| sext.r Rd | | 7 | Rd | 2 | 12 | 6 |
| irqs, print.r ver lic Rd / Rs | | 7 | Rd / Rs | 4 7 | 12 | 6 |
| irqe irqd rtie, brk halt sleepnop, sleepsrf nop, srf | | 7 | 0 7 | 0 | 13 | 6 |
| srf | | 7 | 0 | 1 | 13 | 6 |
| FREE | | 7 | 0 7 | 2 7 | 13 | 6 |
| mov.r Rd Rs | | 7 | Rs | Rd | 14 | 6 |
| mov.32.r Rd Rs | | 7 | Rs | Rd | 15 | 6 |

## 8.3 Structure of 32-bit Instructions




Bit
Nibble
N7
N6
N5
N4
OPE
OPD2
OPC2
OPB2
OPJ
OPI
OPD2
OPC2
OPB2
OPA2
OP2
N3
N2
N1
N0
OPD
OPC
OPB
OPA
OP
FREE
mult sh r Rd Rs Rt #shift[4 0]
shftr 32s r Rd, Rs, #shift[4 0]
shftr 32u r Rd, Rs, #shift[4 0]
shftl 32 r Rd, Rs, #shift[4 0]
rotatel 32 r Rd, Rs, #shift[4 0]
FREE
FREE
FREE
movr2s movs2r, movr2b, movb2r ?d, ?s
tmp r Rs
swap r Rd @(%rO Ra)
Rd = 32 bit
Rs = Rt = 16 bit
mult 32s r Rd Rs, Rt
Rd = 32 bit
Rs = Rt = 16-bit
mult 32u r Rd, Rs Rt
Rd = Rs = 32 bit
Rt = 16 bit
shftr 32s r Rd, Rs, Rt
Rd = Rs = 32-bit
Rt = 16 bit
shftr 32u r Rd, Rs, Rt
Rd = Rs = 32 bit
Rt = 16 bit
shftl 32 r Rd Rs, Rt
Rd = Rs = 32 bit
Rt = 16-bit
rotatel 32 r Rd Rs, Rt
stz r @(Rx, Ra)
stz 8 r @(Rx, Ra)
FREE
div * r Rd Rs, Rt
rem * r Rd Rs, Rt
Rd R(d+1) Rs Rt = 16 bit
Rd = Rs / Rt
R(d+1) = Rs % Rt
dvrem * r Rd Rs Rt
FREE
Rd = DstPtr Rs = SrcPtr Rt = Size
Rd = Rs = Rt = Bytes
blkcp r Rd, Rs Rt
Rd = DstPtr Rs = SrcPtr, Rt = Size
Rd = Rs = Rt = Bytes
blkcps r Rd, Rs Rt
Rd = DstPtr Rs = SrcReg, Rt = Size
Rd = Rt = Bytes
blkst r Rd Rs Rt
Rd = DstPtr Rs = SrcReg, Rt = Size
Rd = Rt = Bytes
blkst 8 r Rd, Rs, Rt
FREE
FREE
srl

## 8.4 Base Instructions

The table below is an alphabetical list of the mnemonics representing all the instructions in XAP4. For each mnemonic there are details of the operands specified in the assembly language and a brief description of the instruction.

There are 3 types of instruction :

- 16-bit only (e.g most .r instructions)
- 32-bit only (e.g most .i instructions)
- 16-bit and 32-bit (commonly used .i instructions)

The 3 privileged modes (NMI, Interrupt, Supervisor) can use all of these instructions. A subset of these instructions are available in User mode. Section 6.7.9 lists the instructions that are not available in User mode.

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| add.c.i | Rd, Rs, #immediate[15:0]s | Add Immediate with Carry<br>Rd = Rs + #imm | 32 |
| add.c.r | Rd, Rs, Rt | Add Register with Carry<br>Rd = Rs + #imm + C | 16 |
| add.i | Rd, Rs, #immediate[15:0]s | Add Immediate<br>Rd = Rs + #imm | 16, 32 |
| add.r | Rd, Rs, Rt | Add Register<br>Rd = Rs + Rt | 16 |
| and.i | Rd, Rs, #immediate[15:0]s | AND Immediate<br>Rd = Rs & #imm | 16, 32 |
| and.r | Rd, Rs, Rt | AND Register<br>Rd = Rs & Rt | 16 |
| bcc (bge.u) | #offset[15:0]s | Branch if Carry Clear | 32 |
| bcs (blt.u) | #offset[15:0]s | Branch if Carry Set | 32 |
| beq | #offset[15:0]s | Branch if Equal | 16, 32 |
| bge.s | #offset[15:0]s | Branch if Greater Than or Equal (Signed) | 16, 32 |
| bgt.s | #offset[15:0]s | Branch if Greater Than (Signed) | 16, 32 |
| bgt.u | #offset[15:0]s | Branch if Greater Than (Unsigned) | 16, 32 |
| ble.s | #offset[15:0]s | Branch if Less Than or Equal (Signed) | 16, 32 |
| ble.u | #offset[15:0]s | Branch if Less Than (Unsiged) | 16, 32 |
| blkcp.r | Rd, Rs, Rt | Block Copy<br>Rd = Destination Pointer<br>Rs = Source Pointer<br>Rt = Size | 32 |

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| | | All arguments in bytes | |
| blkcps.r | Rd, Rs, Rt | Block Copy String<br>Rd = Destination Pointer<br>Rs = Source Pointer<br>Rt = Size<br>All arguments in bytes | 32 |
| blkst.8.r | Rd, Rs, Rt | Block Store (8-bit)<br>Rd = Destination Pointer<br>Rs = Source Register<br>Rt = Size<br>Rd and Rt in bytes | 32 |
| blkst.r | Rd, Rs, Rt | Block Store<br>Rd = Destination Pointer<br>Rs = Source Register<br>Rt = Size<br>Rd and Rt in bytes | 32 |
| blt.s | #offset[15:0]s | Branch if Less Than (Signed) | 16, 32 |
| bmi | #offset[15:0]s | Branch if Minus | 32 |
| bne | #offset[15:0]s | Branch if Not Equal | 16, 32 |
| bpl | #offset[15:0]s | Branch if Plus | 32 |
| bra.a | #address[15:0]u | Branch Absolute | 32 |
| bra.p | #offset[15:0]s | Branch PC Relative | 16, 32 |
| bra.r | Rs | Branch Register | 16 |
| brk | | Break | 16 |
| bsr.a | #address[15:0]u | Branch Subroutine Absolute | 32 |
| bsr.p | #offset[15:0]s | Branch Subroutine Relative | 32 |
| bsr.r | Rs | Branch to Subroutine Register | 16 |
| bvc | #offset[15:0]s | Branch if Overflow Clear | 32 |
| bvs | #offset[15:0]s | Branch if Overflow Set | 32 |
| cmp.8.i | Rs, #immediate[7:0] | Compare Immediate | 16, 32 |
| cmp.8.r | Rs, Rt | Compare Register | 16 |
| cmp.8c.i | Rs, #immediate[7:0] | Compare with Carry Immediate | 32 |
| cmp.8c.r | Rs, Rt | Compare with Carry Register | 16 |
| cmp.8x.i | Rs, #immediate[7:0] | Compare Immediate | 32 |
| cmp.8xc.i | Rs, #immediate[7:0] | Compare with Carry Immediate | 32 |
| cmp.c.i | Rs, #immediate[15:0]s | Compare with Carry Immediate | 32 |
| cmp.c.r | Rs, Rt | Compare with Carry Register | 16 |
| cmp.i | Rs, #immediate[15:0]s | Compare Immediate | 16, 32 |
| cmp.r | Rs, Rt | Compare Register | 16 |
| cmp.x.i | Rs, #immediate[15:0]s | Compare Immediate | 32 |

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| cmp.xc.i | Rs, #immediate[15:0]s | Compare with Carry Immediate | 32 |
| div.32s.i | Rd, Rs, #immediate[15:0]s | 32/16 Divide Immediate (Signed)<br>Rd = {R(s+1), Rs} / #imm | 32 |
| div.32s.r | Rd, Rs, Rt | 32/16 Divide Register (Signed)<br>Rd = {R(s+1), Rs} / Rt | 32 |
| div.32u.i | Rd, Rs, #immediate[15:0]u | 32/16 Divide Immediate (Unsigned)<br>Rd = {R(s+1), Rs} / #imm | 32 |
| div.32u.r | Rd, Rs, Rt | 32/16 Divide Register (Unsigned)<br>Rd = {R(s+1), Rs} / Rt | 32 |
| div.s.i | Rd, Rs, #immediate[15:0]s | Divide Immediate (Signed)<br>Rd = Rs / #imm | 32 |
| div.s.r | Rd, Rs, Rt | Divide Register (Signed)<br>Rd = Rs / Rt | 32 |
| div.u.i | Rd, Rs, #immediate[15:0]u | Divide Immediate (Unsigned)<br>Rd = Rs / #imm | 32 |
| div.u.r | Rd, Rs, Rt | Divide Register (Unsigned)<br>Rd = Rs / Rt | 32 |
| divrem.32s.i | Rd, Rs, #immediate[15:0]s | 32/16 Divide/Remainder Immediate (Signed)<br>Rd = {R(s+1), Rs} / #imm<br>R(d+1) = {R(s+1), Rs} % #imm | 32 |
| divrem.32s.r | Rd, Rs, Rt | 32/16 Divide/Remainder Register (Signed)<br>Rd = {R(s+1), Rs} / Rt<br>R(d+1) = {R(s+1), Rs} % Rt | 32 |
| divrem.32u.i | Rd, Rs, #immediate[15:0]u | 32/16 Divide/Remainder Immediate (Unsigned)<br>Rd = {R(s+1), Rs} / #imm<br>R(d+1) = {R(s+1), Rs} % #imm | 32 |
| divrem.32u.r | Rd, Rs, Rt | 32/16 Divide/Remainder Register (Unsigned)<br>Rd = {R(s+1), Rs} / Rt<br>R(d+1) = {R(s+1), Rs} % Rt | 32 |
| divrem.s.i | Rd, Rs, #immediate[15:0]s | Divide/Remainder Immediate (Signed)<br>Rd = Rs / #imm<br>R(d+1) = Rs % #imm | 32 |
| divrem.s.r | Rd, Rs, Rt | Divide/Remainder Register (Signed)<br>Rd = Rs / Rt | 32 |

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| | | R(d+1) = Rs % Rt | |
| divrem.u.i | Rd, Rs, #immediate[15:0]u | Divide/Remainder Immediate (Unsigned)<br>Rd = Rs / #imm<br>R(d+1) = Rs % #imm | 32 |
| divrem.u.r | Rd, Rs, Rt | Divide/Remainder Register (Unsigned)<br>Rd = Rs / Rt<br>R(d+1) = Rs % Rt | 32 |
| halt | | Halt | 16 |
| irqd | | Disable Interrupts | 16 |
| irqe | | Enable Interrupts | 16 |
| irqs | Rd | Interrupt Status | 16 |
| ld.8z.i | Rd, @(offset[15:0]s, Ra) | Load with Displacement Addressing (8-bit, Zero Extend) | 16, 32 |
| ld.8z.p | Rd, @(offset[15:0]s) | Load with PC-Relative Addressing (8-bit, Zero Extend) | 32 |
| ld.8z.r | Rd, @(Rx, Ra) | Load with Indexed Addressing (8-bit, Zero Extend) | 16 |
| ld.i | Rd, @(offset[15:0]s, Ra) | Load with Displacement Addressing | 16, 32 |
| ld.p | Rd, @(offset[15:0]s) | Load with PC-Relative Addressing | 32 |
| ld.r | Rd, @(Rx, Ra) | Load with Indexed Addressing | 16 |
| lic | Rd | Move processor licence number to Register | 16 |
| mov.i | Rd, #immediate[15:0]s | Move Immediate<br>Rd = #imm | 16, 32 |
| mov.p | Rd, #offset[15:1]s | Move PC Relative Address to Register | 32 |
| mov.32.r | Rd, Rs | Move Accumulator<br>{R(d+1), Rd} = {R(s+1), Rs} | 16 |
| mov.r | Rd, Rs | Move Register<br>Rd = Rs | 16 |
| movb2r | Rd, Bs | Move Breakpoint to Register | 32 |
| movr2b | Bd, Rs | Move Register to Breakpoint | 32 |
| movr2s | Sd, Rs | Move Register to Special | 32 |
| movs2r | Rd, Ss | Move Special to Register | 32 |
| mult.32s.i | Rd, Rs, #immediate[15:0]s | Multiply Immediate (Signed)<br>{R(d+1), Rd} = Rs * #imm | 32 |
| mult.32s.r | Rd, Rs, Rt | Multiply Register (Signed)<br>{R(d+1), Rd} = Rs * Rt | 32 |

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| mult.32u.i | Rd, Rs, #immediate[15:0]u | Multiply Immediate (Unsigned)<br>{R(d+1), Rd} = Rs * #imm | 32 |
| mult.32u.r | Rd, Rs, Rt | Multiply Register (Unsigned)<br>{R(d+1), Rd} = Rs * Rt | 32 |
| mult.i | Rd, Rs, #immediate[15:0]s | Multiply Immediate<br>Rd = bottom 16 bits of (Rs * #imm) | 32 |
| mult.r | Rd, Rs, Rt | Multiply Register<br>Rd = bottom 16 bits of (Rs * Rt) | 16 |
| mult.sh.r | Rd, Rs, Rt, #immediate[4:0]u | Multiply then shift right Register.<br>Rd = (Rs * Rt) >> #imm<br>Signed multiply and shift. | 32 |
| nop | | No Operation | 16 |
| or.i | Rd, Rs, #immediate[15:0]s | OR Immediate<br>Rd = Rs \| #imm | 16, 32 |
| or.r | Rd, Rs, Rt | OR Register<br>Rd = Rs \| Rt | 16 |
| pop | {RegList[3:0]}, #offset[4:1]u | Pop from Stack | 16 |
| pop.ret | {RegList[3:0]}, #offset[4:1]u | Pop from Stack and Return | 16 |
| print.r | Rs | Print Register | 16 |
| push | {RegList[3:0]}, #offset[4:1]u | Push to Stack | 16 |
| rem.32s.i | Rd, Rs, #immediate[15:0]s | 32/16 Remainder Immediate (Signed)<br>Rd = {R(s+1), Rs} % #imm | 32 |
| rem.32s.r | Rd, Rs, Rt | 32/16 Remainder Register (Signed)<br>Rd = {R(s+1), Rs} % Rt | 32 |
| rem.32u.i | Rd, Rs, #immediate[15:0]u | 32/16 Remainder Immediate (Unsigned)<br>Rd = {R(s+1), Rs} % #imm | 32 |
| rem.32u.r | Rd, Rs, Rt | 32/16 Remainder Register (Unsigned)<br>Rd = {R(s+1), Rs} % Rt | 32 |
| rem.s.i | Rd, Rs, #immediate[15:0]s | Remainder Immediate (Signed)<br>Rd = Rs % #imm | 32 |
| rem.s.r | Rd, Rs, Rt | Remainder Register (Signed)<br>Rd = Rs % Rt | 32 |
| rem.u.i | Rd, Rs, #immediate[15:0]u | Remainder Immediate (Unsigned)<br>Rd = Rs % #imm | 32 |
| rem.u.r | Rd, Rs, Rt | Remainder Register (Unsigned)<br>Rd = Rs % Rt | 32 |
| rotatel.32.i | Rd, Rs, #immediate[4:0]u | Rotate Left Immediate (32-bit) | 16 |
| rotatel.32.r | Rd, Rs, Rt | Rotate Left Register (32-bit) | 16 |

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| rotatel.i | Rd, Rs, #immediate[3:0]u | Rotate Left Immediate | 16 |
| rotatel.r | Rd, Rs, Rt | Rotate Left Register | 16 |
| rtie | | Return from Interrupt/Exception | 16 |
| sext.r | Rd | Sign extend. Set Rd[15:8] = Rd[7] | 16 |
| shiftl.32.i | Rd, Rs, #immediate[4:0]u | Shift Left Immediate (32-bit) | 16 |
| shiftl.32.r | Rd, Rs, Rt | Shift Left Register (32-bit) | 16 |
| shiftl.i | Rd, Rs, #immediate[3:0]u | Shift Left Immediate | 16 |
| shiftl.r | Rd, Rs, Rt | Shift Left Register | 16 |
| shiftr.32s.i | Rd, Rs, #immediate[4:0]u | Shift Right Immediate (32-bit Signed) | 16 |
| shiftr.32s.r | Rd, Rs, Rt | Shift Right Register (32-bit Signed) | 16 |
| shiftr.s.i | Rd, Rs, #immediate[3:0]u | Shift Right Immediate (Signed) | 16 |
| shiftr.s.r | Rd, Rs, Rt | Shift Right Register (Signed) | 16 |
| shiftr.32u.i | Rd, Rs, #immediate[4:0]u | Shift Right Immediate (32-bit Unsigned) | 16 |
| shiftr.32u.r | Rd, Rs, Rt | Shift Right Register (32-bit Unsigned) | 16 |
| shiftr.u.i | Rd, Rs, #immediate[3:0]u | Shift Right Immediate (Unsigned) | 16 |
| shiftr.u.r | Rd, Rs, Rt | Shift Right Register (Unsigned) | 16 |
| sif | | SIF | 16, 32 |
| sleepnop | | Sleep | 16 |
| sleepsif | | Sleep and Allow SIF | 16 |
| st.8.i | Rs, @(offset[15:0]s, Ra) | Store with Displacement Addressing (8-bit) | 16, 32 |
| st.8.p | Rs, @(offset[15:0]s) | Store with PC-Relative Addressing (8-bit) | 32 |
| st.8.r | Rs, @(Rx, Ra) | Store with Indexed Addressing (8-bit) | 16 |
| st.i | Rs, @(offset[15:0]s, Ra) | Store with Displacement Addressing | 16, 32 |
| st.p | Rs, @(offset[15:0]s) | Store with PC-Relative Addressing | 32 |
| st.r | Rs, @(Rx, Ra) | Store with Indexed Addressing | 16 |
| stz.8.i | @(offset[15:0]s, Ra) | Store Zero with Displacement Addressing (8-bit) | 32 |
| stz.8.p | @(offset[15:0]s) | Store Zero with PC-Relative Addressing (8-bit) | 32 |
| stz.8.r | @(Rx, Ra) | Store Zero with Displacement Addressing (8-bit) | 32 |
| stz.i | @(offset[15:0]s, Ra) | Store Zero with Displacement Addressing | 16, 32 |

| *Mnemonic* | *Operands* | *Operation* | *Available Op* |
|---|---|---|---|
| stz.p | @(offset[15:0]s) | Store Zero with PC-Relative Addressing | 32 |
| stz.r | @(Rx, Ra) | Store Zero with Displacement Addressing | 32 |
| sub.c.r | Rd, Rs, Rt | Subtract Register with Carry<br>Rd = Rs – Rt - C | 16 |
| sub.r | Rd, Rs, Rt | Subtract Register<br>Rd = Rs - Rt | 16 |
| sub.xc.i | Rd, Rs, #immediate[15:0]s | Subtract Immediate with Carry (Reverse Order)<br>Rd = #imm – Rs - C | 32 |
| sub.x.i | Rd, Rs, #immediate[15:0]s | Subtract Immediate (Reverse Order)<br>Rd = #imm - Rs | 32 |
| swap.i | Rd, @(0, Ra) | Swap Register with Memory | 32 |
| trap.i | #immediate[15:0]s | Trap Immediate | 32 |
| trap.r | Rs | Trap Register | 32 |
| ver | Rd | Move processor version number to Register | 16 |
| xor.i | Rd, Rs, #immediate[15:0]s | XOR Immediate<br>Rd = Rs ^ #imm | 32 |
| xor.r | Rd, Rs, Rt | XOR Register<br>Rd = Rs ^ Rt | 16 |

## 8.5 Aliased Instructions

Several instructions have aliases (alternate names) that may be written in an assembler source file and will be translated into the corresponding real instruction by the assembler.

| *Alias Mnemonic* | *Real Instruction* | *Min Op* |
|---|---|---|
| bra.p.2 label | bra.p label | 16 |
| bra.p.4 label | bra.p label | 32 |
| blt.u label | bcs label | 16 |
| bge.u label | bcc label | 16 |

## 8.6 Flag Usage Conventions

The four main arithmetic flags (Z, N, C, V) are set by various instructions and also as defined by the function calling convention. These flags are set according to the rules defined in 5.7. They are normally set for the resultant value in Rd.

The normal convention is that if there has been an overflow and Rd is updated, then it should be set to 0.

The logic instructions (and.*, or.*, xor.*, mov.i, mov.r) update the Z and N flags.

The load and swap instructions update the Z and N flags.

The store instructions do not update any flags.

The block instructions do not update any flags.

The arithmetic instructions (add, subtract, compare) update the Z, N, C and V flags. The carry versions of these instructions (add.c.*, sub.*c.*, cmp.*c.*) set the Z flag in a special way :

- If (result=0) and (Z=1), set Z=1. Otherwise set Z=0.
- This means that if (Z=0) before the instruction, then Z will always be 0 after the instruction (regardless of the instruction result).
- This is useful for 32-bit arithmetic.

The multiply instructions update the Z and N flags. mult.sh.r also updates the V flag.

The div.* and rem.* instructions update the Z and N flags, but the divrem.* instructions do not. The signed divide instructions update the V flag. The unsigned divide instructions update the C flag. The C and V flags are set identically for the div.*, rem.* and divrem.* instructions.

The branch instructions (conditional and unconditional) do not update any flags.

The push, pop and pop.ret instruction do not update any flags.

The shift and rotate instructions update the Z, N and C flags.

## 8.7 Interpretation of Immediates

### *8.7.1 Immediates*

- Immediate instructions (Type = .i) have a 16-bit immediate, #immediate[15:0].

The immediate range is :

| *Immediate Range* | *Dec* | | *Hex* | |
|---|---|---|---|---|
| | *Min* | *Max* | *Min* | *Max* |
| *Signed* | -32768 | 32767 | 0x8000 | 0x7FFF |
| *Unsigned* | 0 | 65535 | 0x0000 | 0xFFFF |
| *Valid Assembler* | -32768 | 65535 | 0x0000 | 0xFFFF |

It can represent numbers such as :

- 0x1234

- 0x4321

### *8.7.2 Addresses*

All instructions are 16 or 32-bit. They are arranged to start on 2byte boundaries (i.e at even addresses). There are several instructions that pass 16-bit addresses or offsets, on 2byte boundaries :

- bra.a, bsr.a // Both use unsigned absolute address[15:1]. address[0] = 0.
- bra.p, bsr.p // Both use signed PC-relative offset[15:1]. offset[0] = 0.

However, mov.p is different as it is used to generate addresses for code and data objects. Data objects can start at odd addresses, so mov.p uses a full 16-bit offset.

- mov.p // adds (PC + offset[15:0]) to Rd

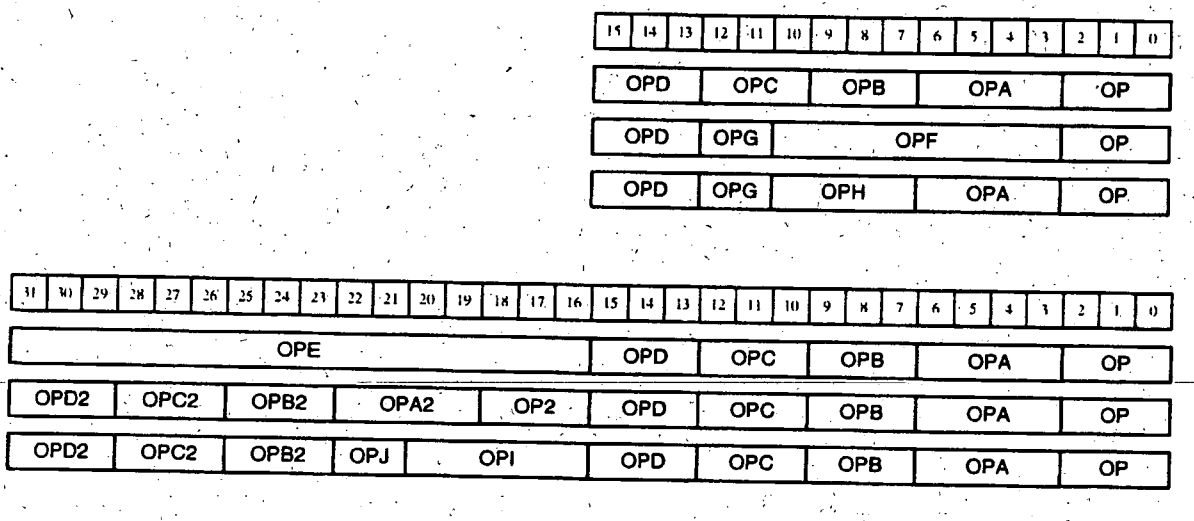
## 8.8 Instruction Opcodes

XAP4 programs are normally stored in 16-bit memory. Of course the XAP4 can also be used with 8-bit memory (with a different MMU). However, this section will assume the more common case where 16-bit memory is used.

The primary opcode OP is bits[2:0] of the instruction. OP values are interpreted as follows :

- 0 – 6 16-bit instruction Single fetch.
- 7 32-bit instruction Double fetch

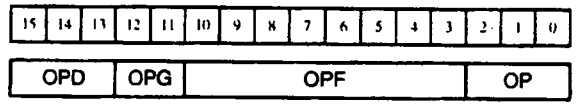
The 16 and 32-bit instruction formats are shown below. The common field definitions between the 16 and 32-bit instructions enable a simple direct decode, without any need to convert from 16 to 32-bit instructions (or vice-versa).




The following sections define the XAP4 instruction set.

Each instruction indicates the number of clock cycles required. This assumes that the memory is 16-bit single-cycle and that the hardware architecture is XAP4a (2-stage pipeline).

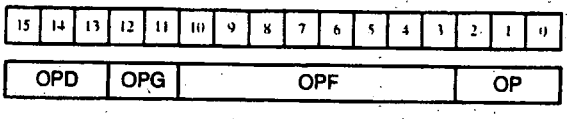
### 8.8.1 OP = 0 – 3, 16-bit instructions




| *OPD* | *OPG* | *OPF* | *OP* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|
| Ra | Rd<br>0 = R0<br>1 = R1<br>2 = R2<br>3 = R3 | offset[8:1]u | 0 | ld.i Rd, @(offset, Ra) | Z, N | 2 |
| Ra | Rd<br>0 = R4<br>1 = R5<br>2 = R6 | offset[8:1]u | 1 | ld.i Rd, @(offset, Ra) | Z, N | 2 |
| Ra | 3 | offset[7:0]u | 1 | ld.8z.i R0, @(offset, Ra) | Z, N | 2 |
| Ra | Rs<br>0 = R0<br>1 = R1<br>2 = R2<br>3 = R3 | offset[8:1]u | 2 | st.i Rs, @(offset, Ra) | | 2 |
| Ra | Rs<br>0 = R4<br>1 = R5<br>2 = R6 | offset[8:1]u | 3 | st.i Rs, @(offset, Ra) | | 2 |
| Ra | 3 | offset[7:0]u | 3 | st.8.i R0, @(offset, Ra) | | 2 |

These instructions are all available in 32-bit form as well.

### 8.8.2 OP = 4, 16-bit instructions



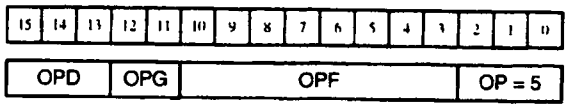

For most of the following instruction decodes, #imm represents the immediate used in the assembler code and the argument in the instruction. In some more complicated cases, #imm2 is the immediate in the assembler code, while #imm is still the argument in the instruction.

| *OPD* | *OPG* | *OPF* | *OP* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|
| Rd<br>0 - 6 | 0 | imm[7:0]u | 4 | mov.i Rd, #imm | Z, N | 1 |
| 7 | 0 | imm[8:1]u | 4 | add.i R0, R7, #imm | Z, N, C,V | 1 |
| Rs<br>0 - 6 | 1 | imm[7:0]u | 4 | cmp.i Rs, #imm | Z, N, C,V | 1 |
| 7 | 1 | imm[8:1]u | 4 | add.i R1, R7, #imm | Z, N, C,V | 1 |
| Rd<br>0 - 6 | 2 | imm[7:0]u | 4 | and.i Rd, Rd, #imm | Z, N | 1 |
| 7 | 2 | imm[8:1]u | 4 | add.i R2, R7, #imm | Z, N, C,V | 1 |
| Rd<br>0 - 6 | 3 | imm[7:0]u | 4 | add.i Rd, Rd, #imm | Z, N, C,V | 1 |
| 7 | 3 | imm[8:1]u | 4 | add.i R7, R7, #imm | Z, N, C,V | 1 |

These instructions are all available in 32-bit form as well.

### *8.8.3 OP = 5, 16-bit instructions*

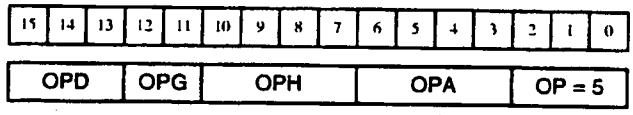
#### 8.8.3.1 OP = 5, Branches




| *OPD* | *OPG* | *OPF* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|
| 0 | 0 | offset[8:1]s | beq offset | | 1 / 2 |
| 0 | 1 | offset[8:1]s | bne offset | | 1 / 2 |
| 0 | 2 | offset[8:1]s | ble.s offset | | 1 / 2 |
| 0 | 3 | offset[8:1]s | bgt.s offset | | 1 / 2 |
| 1 | 0 | offset[8:1]s | blt.s offset | | 1 / 2 |
| 1 | 1 | offset[8:1]s | bge.s offset | | 1 / 2 |
| 1 | 2 | offset[8:1]s | ble.u offset | | 1 / 2 |
| 1 | 3 | offset[8:1]s | bgt.u offset | | 1 / 2 |
| 2 | 0 - 3 | offset[8:1]s | FREE | | |
| 3 | 0 | offset[8:1]s | bra.p offset | | 2 |

The number of clock cycles = a / b. a = branch not taken. b = branch taken.
These instructions are all available in 32-bit form as well.

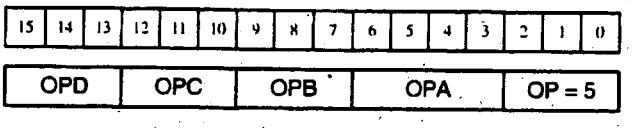
#### 8.8.3.2 OP = 5, Push, Pop




| *OPD* | *OPG* | *OPH* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|
| 3 | 1 | reglist[3:0] | offset[4:1]u | push {reglist}, #offset | | 2 – 8 |
| 3 | 2 | reglist[3:0] | offset[4:1]u | pop {reglist}, #offset | | 2 – 9 |
| 3 | 3 | reglist[3:0] | offset[4:1]u | pop.ret {reglist}, #offset | | 4 – 11 |

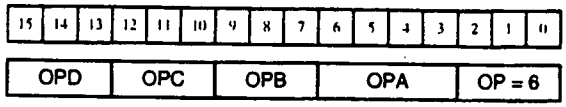
The number of clock cycles depends upon the number of registers in reglist[3:0].

### 8.8.3.3 OP = 5, 16-bit Shifts




| *OPD* | *OPC* | *OPB* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|
| 4 | Rs<br>0 - 6 | Rd<br>0 - 6 | imm[3:0]u | shiftr.s.i Rd, Rs, #imm | Z, N, C | 1 |
| 4 | 7 | Rd<br>0 - 6 | imm[3:0]u | or.i Rd, Rd, #imm | Z, N | 1 |
| 4 | Rs<br>0 - 6 | 7 | imm[3:0]u | cmp.8.i Rs, #imm | Z, N, C,V | 1 |
| 5 | Rs<br>0 – 6 | Rd<br>0 – 6 | imm[3:0]u | shiftr.u.i Rd, Rs, #imm | Z, N, C | 1 |
| 5 | 7 | offset[7:5] | offset[4:1]u | stz.i @(offset, R7) | | 2 |
| 5 | Ra<br>0 - 6 | 7 | offset[4:1]u | stz.i @(offset, Ra) | | 2 |
| 6 | Rs<br>0 - 6 | Rd<br>0 – 6 | imm[3:0]u | shiftl.i Rd, Rs, #imm | Z, N, C | 1 |
| 6 | 7 | Rd<br>0 - 6 | imm[3:0]u | mov.i Rd, (#imm – 16) | Z, N | 1 |
| 6 | Rs<br>0 - 6 | 7 | imm[3:0]u | cmp.i Rs, (#imm – 16) | Z, N, C,V | 1 |
| 7 | Rs<br>0 - 6 | Rd<br>0 - 6 | imm[3:0]u | rotatel.i Rd, Rs, #imm | Z, N, C | 1 |
| 7 | 7 | imm[7:5] | imm[4:1]u | add.i R7, R7, (#imm – 256) | Z, N, C,V | 1 |
| 7 | Rd<br>0 - 6 | 7 | imm[3:0]u | add.i Rd, Rd, (#imm – 16) | Z, N, C,V | 1 |

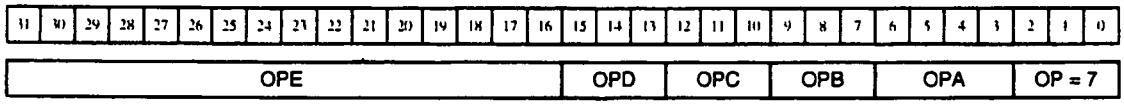
### 8.8.4 OP = 6, 16-bit instructions




| *OPD* | *OPC* | *OPB* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|
| Rd | Ra | Rx | 0 | ld.r Rd, @(Rx, Ra) | Z, N | 2 |
| Rd | Ra | Rx | 1 | ld.8z.r Rd, @(Rx, Ra) | Z, N | 2 |
| Rs | Ra | Rx | 2 | st.r Rd, @(Rx, Ra) | | 2 |
| Rs | Ra | Rx | 3 | st.8.r Rd, @(Rx, Ra) | | 2 |
| Rd<br>0 – 6 | Rs<br>0 - 6 | Rt<br>0 - 6 | 4 | and.r Rd, Rs, Rt | Z, N | 1 |
| 7 | Rs | Rt | 4 | cmp.8.r Rs, Rt | Z, N, C,V | 1 |
| Rd<br>0 - 6 | Rs<br>0 - 6 | Rt<br>0 - 6 | 5 | or.r Rd, Rs, Rt | Z, N | 1 |
| 7 | Rs | Rt | 5 | cmp.8c.r Rs, Rt | Z, N, C,V | 1 |
| Rd<br>0 - 6 | Rs<br>0 – 6 | Rt<br>0 - 6 | 6 | xor.r Rd, Rs, Rt | Z, N | 1 |
| 7 | Rs | Rt | 6 | cmp.r Rs, Rt | Z, N, C,V | 1 |
| Rd<br>0 - 6 | Rs<br>0 - 6 | Rt<br>0 - 6 | 7 | mult.r Rd, Rs, Rt | Z, N | 16 |
| 7 | Rs | Rt | 7 | cmp.c.r Rs, Rt | Z, N, C,V | 1 |
| Rd | Rs | Rt | 8 | add.r Rd, Rs, Rt | Z, N, C,V | 1 |
| Rd | Rs | Rt | 9 | add.c.r Rd, Rs, Rt | Z, N, C,V | 1 |
| Rd | Rs | Rt | 10 | sub.r Rd, Rs, Rt | Z, N, C,V | 1 |
| Rd | Rs | Rt | 11 | sub.c.r Rd, Rs, Rt | Z, N, C,V | 1 |
| Rd<br>0 - 6 | Rs<br>0 - 6 | Rt<br>0 - 6 | 12 | shiftr.s.r Rd, Rs, Rt | Z, N, C | 1 |
| 7 | Ra | 0 | 12 | bra.r Ra | | 2 |
| 7 | Ra<br>Not 6 | 1 | 12 | bsr.r Ra | | 3 |
| 7 | Rd | 2 | 12 | sext.r Rd | Z, N | 1 |
| 7 | 0 - 7 | 3 | 12 | FREE | | |
| 7 | Rd | 4 | 12 | irqs Rd | Z, N | 1 |
| 7 | Rs | 5 | 12 | print.r Rs | | 1 |
| 7 | Rd | 6 | 12 | ver Rd | | 1 |
| 7 | Rd | 7 | 12 | lic Rd | | 1 |
| Rd<br>0 - 6 | Rs<br>0 – 6 | Rt<br>0 - 6 | 13 | shiftr.u.r Rd, Rs, Rt | Z, N, C | 1 |

| *OPD* | *OPC* | *OPB* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|
| 7 | 0 | 0 | 13 | irqe | | 1 |
| 7 | 1 | 0 | 13 | irqd | | 1 |
| 7 | 2 | 0 | 13 | rtie | Z, N, C,V | 2 |
| 7 | 3 | 0 | 13 | brk | | 1 |
| 7 | 4 | 0 | 13 | halt | | 1 |
| 7 | 5 | 0 | 13 | sleepnop | | 1 |
| 7 | 6 | 0 | 13 | sleepsif | | 1 |
| 7 | 7 | 0 | 13 | nop | | 1 |
| 7 | 0 | 1 | 13 | sif | | 4 |
| 7 | 1 - 7 | 1 | 13 | FREE | | |
| 7 | | 2 - 7 | 13 | FREE | | |
| Rd<br>0 - 6 | Rs<br>0 – 6 | Rt<br>0 - 6 | 14 | shiftl.r Rd, Rs, Rt | Z, N, C | 1 |
| 7 | Rs | Rd | 14 | mov.r Rd, Rs | Z, N | 1 |
| Rd<br>0 - 6 | Rs<br>0 - 6 | Rt<br>0 - 6 | 15 | rotatel.r Rd, Rs, Rt | Z, N, C | 1 |
| 7 | Rs<br>0 - 5 | Rd<br>0 - 5 | 15 | mov.32.r Rd, Rs | Z, N | 1 |

### 8.8.5 OP = 7, 32-bit RRI instructions

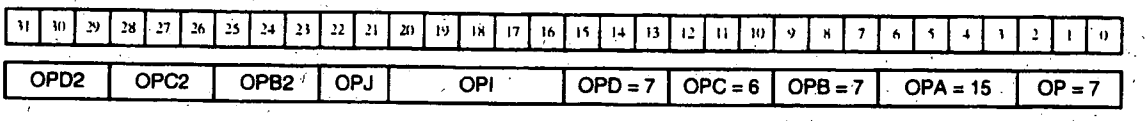



| *OPE* | *OPD* | *OPC* | *OPB* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|---|
| offset[15:0]s | Rd | Ra | 0 | 0 | ld.i Rd, @(offset, Ra) | Z, N | 3 |
| offset[15:0]s | Rd | Ra | 1 | 0 | ld.8z.i Rd, @(offset, Ra) | Z, N | 3 |
| offset[15:0]s | Rs | Ra | 2 | 0 | st.i Rs, @(offset, Ra) | | 3 |
| offset[15:0]s | Rs | Ra | 3 | 0 | st.8.i Rs, @(offset, Ra) | | 3 |
| imm[15:0] | Rd | Rs | 4 | 0 | and.i Rd, Rs, #imm | Z, N | 2 |
| imm[15:0] | Rd | Rs | 5 | 0 | or.i Rd, Rs, #imm | Z, N | 2 |
| imm[15:0] | Rd | Rs | 6 | 0 | xor.i Rd, Rs, #imm | Z, N | 2 |
| imm[15:0]s | Rd | Rs | 7 | 0 | mult.i Rd, Rs, #imm | Z, N | 17 |
| imm[15:0]s | Rd | Rs | 0 | 1 | add.i Rd, Rs, #imm | Z, N, C,V | 2 |
| imm[15:0]s | Rd | Rs | 1 | 1 | add.c.i Rd, Rs, #imm | Z, N, C,V | 2 |
| imm[15:0]s | Rd | Rs | 2 | 1 | sub.x.i Rd, Rs, #imm | Z, N, C,V | 2 |
| imm[15:0]s | Rd | Rs | 3 | 1 | sub.xc.i Rd, Rs, #imm | Z, N, C,V | 2 |
| imm[15:0]s | S, C, O params<br>0<br>1<br>2<br>3<br>4<br>5<br>6<br>7 | Rs | 4 | 1 | <br><br>cmp.8.i<br>cmp.8x.i<br>cmp.8c.i<br>cmp.8xc.i<br>cmp.i<br>cmp.x.i<br>cmp.c.i<br>cmp.xc.i Rs, #imm | Z, N, C,V | 2 |
| offset[15:0]s | 0 | Ra | 5 | 1 | stz.i @(offset, Ra) | | 3 |
| offset[15:0]s | 1 | Ra | 5 | 1 | stz.8.i @(offset, Ra) | | 3 |
| offset[15:0]s | 2 | 0 | 5 | 1 | stz.p @(offset)<br>(PC relative) | | 3 |
| offset[15:0]s | 3 | 0 | 5 | 1 | stz.8.p @(offset)<br>(PC relative) | | 3 |
| | 4 – 7 | | 5 | 1 | FREE | | |
| imm[15:0]s | Rd | 0 | 6 | 1 | mov.i Rd, #imm | Z, N | 2 |
| offset[15:0]s | Rd | 1 | 6 | 1 | mov.p Rd, #offset<br>(PC relative) | | 2 |
| | | 2 - 3 | 6 | 1 | FREE | | |

| *OPE* | *OPD* | *OPC* | *OPB* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|---|
| offset[15:0]s | Rd | 4 | 6 | 1 | ld.p Rd, @(offset)<br>(PC relative) | Z, N | 3 |
| offset[15:0]s | Rd | 5 | 6 | 1 | ld.8z.p Rd, @(offset)<br>(PC relative) | Z, N | 3 |
| offset[15:0]s | Rs | 6 | 6 | 1 | st.p Rs, @(offset)<br>(PC relative) | | 3 |
| offset[15:0]s | Rs | 7 | 6 | 1 | st.8.p Rs, @(offset)<br>(PC relative) | | 3 |
| | | | 7 | 1 | FREE | | |
| | | | 0 - 1 | 2 | FREE | | |
| imm[15:0]s | Rd<br>0 - 5 | Rs | 2 | 2 | mult.32s.i Rd, Rs, #imm<br>{R(d+1), Rd}= Rs * #imm | Z, N | 17 |
| imm[15:0]u | Rd<br>0 - 5 | Rs | 3 | 2 | mult.32u.i Rd, Rs, #imm<br>{R(d+1), Rd}= Rs * #imm | Z, N | 17 |
| offset[15:1]s, 0 | 0 = beq<br>1 = bne<br>2 = ble.s<br>3 = bgt.s<br>4 = blt.s<br>5 = bge.s<br>6 = ble.u<br>7 = bgt.u | 0 | 4 | 2 | ConditionalBranch offset<br>(PC relative) | | 2 / 3 |
| offset[15:1]s, 0 | 0 = bcs<br>(blt.u)<br>1 = bcc<br>(bge.u)<br>2 = bvs<br>3 = bvc<br>4 = bmi<br>5 = bpl | 0 | 5 | 2 | ConditionalBranch offset<br>(PC relative) | | 2 / 3 |
| | 6 - 7 | | 5 | 2 | FREE | | |
| address[15:1], 0 | 0 | 0 | 6 | 2 | bra.a address | | 3 |
| address[15:1], 0 | 1 | 0 | 6 | 2 | bsr.a address | | 4 |
| offset[15:1]s, 0 | 2 | 0 | 6 | 2 | bra.p offset<br>(PC relative) | | 3 |
| offset[15:1]s, 0 | 3 | 0 | 6 | 2 | bsr.p offset<br>(PC relative) | | 4 |
| imm[15:0] | 4 | 0 | 6 | 2 | trap.i #imm | | 6 |
| | 5 - 7 | 0 | 6 | 2 | FREE | | |
| | | | 7 | 2 | FREE | | |

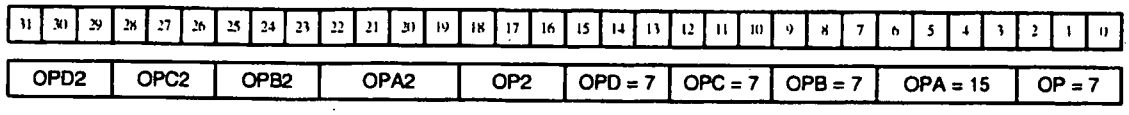
| *OPE* | *OPD* | *OPC* | *OPB* | *OPA* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|---|
| imm[15:0]s | Rd | Rs | 0 | 3 | div.s.i Rd, Rs, #imm | Z, N, V | 20 |
| imm[15:0]u | Rd | Rs | 1 | 3 | div.u.i Rd, Rs, #imm | Z, N, C | 18 |
| imm[15:0]s | Rd | Rs<br>0 - 5 | 2 | 3 | div.32s.i Rd, Rs, #imm<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, V | 20 |
| imm[15:0]u | Rd | Rs<br>0 - 5 | 3 | 3 | div.32u.i Rd, Rs, #imm<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 18 |
| imm[15:0]s | Rd | Rs | 4 | 3 | rem.s.i Rd, Rs, #imm | Z, N, V | 20 |
| imm[15:0]u | Rd | Rs | 5 | 3 | rem.u.i Rd, Rs, #imm | Z, N, C | 18 |
| imm[15:0]s | Rd | Rs<br>0 - 5 | 6 | 3 | rem.32s.i Rd, Rs, #imm<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, V | 20 |
| imm[15:0]u | Rd | Rs<br>0 - 5 | 7 | 3 | rem.32u.i Rd, Rs, #imm<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 18 |
| imm[15:0]s | Rd<br>0 - 5 | Rs | 0 | 4 | divrem.s.i Rd, Rs, #imm<br>Rd = Rs / #imm<br>R(d+1) = Rs % #imm | V | 20 |
| imm[15:0]u | Rd<br>0 - 5 | Rs | 1 | 4 | divrem.u.i Rd, Rs, #imm<br>Rd = Rs / #imm<br>R(d+1) = Rs % #imm | C | 18 |
| imm[15:0]s | Rd<br>0 - 5 | Rs<br>0 - 5 | 2 | 4 | divrem.32s.i Rd, Rs, #imm<br>Rs = 32-bit = {R(s+1), Rs}<br>Rd = Rs / #imm<br>R(d+1) = Rs % #imm | V | 20 |
| imm[15:0]u | Rd<br>0 - 5 | Rs<br>0 - 5 | 3 | 4 | divrem.32u.i Rd, Rs, #imm<br>Rs = 32-bit = {R(s+1), Rs}<br>Rd = Rs / #imm<br>R(d+1) = Rs % #imm | C | 18 |
| | | | 4 - 7 | 4 | FREE | | |
| | | | | 5 - 14 | FREE | | |
| | | 6 | 7 | 15 | 32-bit RRRI instructions | | |
| | | 7 | 7 | 15 | 32-bit RRR instructions | | |

### 8.8.6 OP = 7, 32-bit RRRI instructions




| *OPD2* | *OPC2* | *OPB2* | *OPJ* | *OPI* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|---|
| Rd | Rs | Rt | 0 | shift[4:0] | mult.sh.r Rd, Rs, Rt, #shift<br>Rd = (Rs * Rt) >> #shift | Z, N, V | 18 |
| Rd<br>0 - 5 | Rs<br>0 – 5 | 0 | 1 | shift[4:0] | shiftr.32s.i Rd, Rs, #imm<br>Rd = 32-bit = {R(d+1), Rd}<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 2 |
| Rd<br>0 – 5 | Rs<br>0 – 5 | 1 | 1 | shift[4:0] | shiftr.32u.i Rd, Rs, #imm<br>Rd = 32-bit = {R(d+1), Rd}<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 2 |
| Rd<br>0 – 5 | Rs<br>0 – 5 | 2 | 1 | shift[4:0] | shiftl.32.i Rd, Rs, #imm<br>Rd = 32-bit = {R(d+1), Rd}<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 2 |
| Rd<br>0 – 5 | Rs<br>0 - 5 | 3 | 1 | shift[4:0] | rotatel.32.i Rd, Rs, #imm<br>Rd = 32-bit = {R(d+1), Rd}<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 2 |
| | | 4 - 7 | 1 | | FREE | | |
| | | | 2 - 3 | | FREE | | |

### 8.8.7 OP = 7, 32-bit RRR instructions




| *OPD2* | *OPC2* | *OPB2* | *OPA2* | *OP2* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|---|
| Sd | Rs | 0 | 0 | 0 | movr2s Sd, Rs | | 2 |
| Rd | Ss | 0 | 1 | 0 | movs2r Rd, Ss | Z, N | 2 |
| Bd | Rs | 0 | 2 | 0 | movr2b Bd, Rs | | 2 |
| Rd | Bs | 0 | 3 | 0 | movb2r Rd, Bs | Z, N | 2 |
| 0 | Rs | 0 | 4 | 0 | trap.r Rs | | 6 |
| Rd | Ra | 0 | 5 | 0 | swap.i Rd, @(0, Ra) | Z, N Set on new Rd value | 5 |
| Rd 0 - 5 | Rs | Rt | 6 | 0 | mult.32s.r Rd, Rs, Rt {R(d+1), Rd}= Rs * Rt | Z, N | 17 |
| Rd 0 - 5 | Rs | Rt | 7 | 0 | mult.32u.r Rd, Rs, Rt {R(d+1), Rd}= Rs * Rt | Z, N | 17 |
| Rd 0 – 5 | Rs 0 – 5 | Rt | 8 | 0 | shiftr.32s.r Rd, Rs, Rt Rd = 32-bit = {R(d+1), Rd} Rs = 32-bit = {R(s+1), Rs} Use Rt[4:0] | Z, N, C | 2 |
| Rd 0 – 5 | Rs 0 – 5 | Rt | 9 | 0 | shiftr.32u.r Rd, Rs, Rt Rd = 32-bit = {R(d+1), Rd} Rs = 32-bit = {R(s+1), Rs} Use Rt[4:0] | Z, N, C | 2 |
| Rd 0 – 5 | Rs 0 – 5 | Rt | 10 | 0 | shiftl.32.r Rd, Rs, Rt Rd = 32-bit = {R(d+1), Rd} Rs = 32-bit = {R(s+1), Rs} Use Rt[4:0] | Z, N, C | 2 |
| Rd 0 – 5 | Rs 0 - 5 | Rt | 11 | 0 | rotatel.32.r Rd, Rs, Rt Rd = 32-bit = {R(d+1), Rd} Rs = 32-bit = {R(s+1), Rs} Use Rt[4:0] | Z, N, C | 2 |
| 0 | Ra | Rx | 12 | 0 | stz.r @(Rx, Ra) | | 3 |
| 0 | Ra | Rx | 13 | 0 | stz.8.r @(Rx, Ra) | | 3 |
| | | | 14 - 15 | 0 | FREE | | |

| *OPD2* | *OPC2* | *OPB2* | *OPA2* | *OP2* | *Assembler* | *Flags Updated* | *Cycles* |
|---|---|---|---|---|---|---|---|
| Rd | Rs | Rt | 0 | 1 | div.s.r Rd, Rs, Rt | Z, N, V | 20 |
| Rd | Rs | Rt | 1 | 1 | div.u.r Rd, Rs, Rt | Z, N, C | 18 |
| Rd | Rs<br>0 - 5 | Rt | 2 | 1 | div.32s.r Rd, Rs, Rt<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, V | 20 |
| Rd | Rs<br>0 - 5 | Rt | 3 | 1 | div.32u.r Rd, Rs, Rt<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 18 |
| Rd | Rs | Rt | 4 | 1 | rem.s.r Rd, Rs, Rt | Z, N, V | 20 |
| Rd | Rs | Rt | 5 | 1 | rem.u.r Rd, Rs, Rt | Z, N, C | 18 |
| Rd | Rs<br>0 - 5 | Rt | 6 | 1 | rem.32s.r Rd, Rs, Rt<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, V | 20 |
| Rd | Rs<br>0 - 5 | Rt | 7 | 1 | rem.32u.r Rd, Rs, Rt<br>Rs = 32-bit = {R(s+1), Rs} | Z, N, C | 18 |
| Rd<br>0 - 5 | Rs | Rt | 8 | 1 | divrem.s.r Rd, Rs, Rt<br>Rd = Rs / Rt<br>R(d+1) = Rs % Rt | V | 20 |
| Rd<br>0 - 5 | Rs | Rt | 9 | 1 | divrem.u.r Rd, Rs, Rt<br>Rd = Rs / Rt<br>R(d+1) = Rs % Rt | C | 18 |
| Rd<br>0 - 5 | Rs<br>0 - 5 | Rt | 10 | 1 | divrem.32s.r Rd, Rs, Rt<br>Rs = 32-bit = {R(s+1), Rs}<br>Rd = Rs / Rt<br>R(d+1) = Rs % Rt | V | 20 |
| Rd<br>0 - 5 | Rs<br>0 - 5 | Rt | 11 | 1 | divrem.32u.r Rd, Rs, Rt<br>Rs = 32-bit = {R(s+1), Rs}<br>Rd = Rs / Rt<br>R(d+1) = Rs % Rt | C | 18 |
| | | | 12 – 15 | 1 | FREE | | |
| Rd | Rs | Rt | 0 | 2 | blkcp.r Rd, Rs, Rt | | 2n + 8 |
| Rd | Rs | Rt | 1 | 2 | blkcps.r Rd, Rs, Rt | | 2n + 8 |
| Rd | Rs | Rt | 2 | 2 | blkst.r Rd, Rs, Rt | | n + 9 |
| Rd | Rs | Rt | 3 | 2 | blkst.8.r Rd, Rs, Rt | | n + 9 |
| | | | 4 - 15 | 2 | FREE | | |
| | | | | 3 - 6 | FREE | | |
| | | | 0 - 14 | 7 | FREE | | |
| 7 | 7 | 7 | 15 | 7 | sif | | 5 |

The number of clock cycles for the blk* instructions is given in terms of n. n is the number of writes (which could be 8 or 16-bit) executed by the instruction. For a given value of Rt (number of bytes) n will be bigger for unaligned than aligned data.

## 8.9 Instruction Formats

### *8.9.1 Unconditional Branch Instructions*

bra.a, bra.p, bra.p.2, bra.p.4, bra.r

bsr.a, bsr.p, bsr.r

Two unconditional branches are provided :

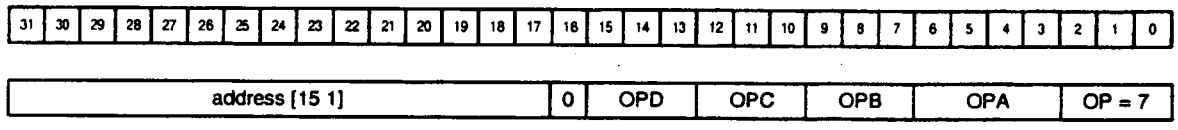
- bra (Branch) will change the program counter to the given argument.
- bsr (Branch Subroutine) will change the program counter and save the address of the instruction after the branch into the link register, R6.

The bra.* and bsr.* instructions do not update any flags.

The fixed length instructions (bra.p.2, bra.p.4) may be useful for jump tables as they force the assembler to generate 2-byte (16-bit) or 4-byte (32-bit) instructions.

#### 8.9.1.1 Absolute

As instructions must start on a 16-bit word boundary, the compiler should only generate even branch addresses. The instruction can only represent even addresses.

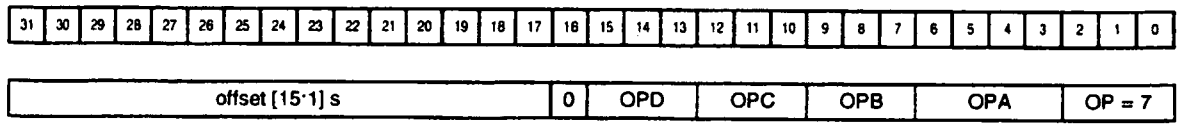



```
bra.a   label       // Assembler calculates address[15:0] label
        bra.a   label129    // Assembler calculates address[15:0] label129
        bsr.a   label35     // Assembler calculates address[15:0] label35
```

#### 8.9.1.2 PC Relative

This branches to an offset address from the Program Counter.

As with the absolute mode, the compiler should only generate even branch offsets. The instruction can only represent even offsets.

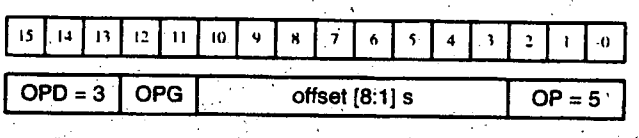



```
bra.p   label       // Assembler calculates address[15:0] label
        bra.p   label32     // Assembler calculates address[15:0] label32
        bsr.p   label782    // Assembler calculates address[15:0] label782
```

Sometimes the compiler wants to force the bra.p to be 2byte or 4byte instructions. In these cases it will generate the mnemonics :

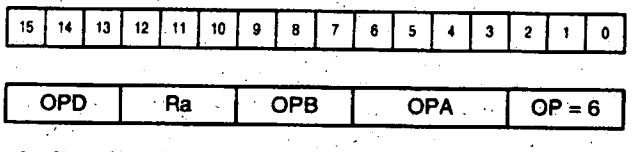
```
bra.p.2     // 2byte, 16-bit instruction
bra.p.4     // 4byte, 32-bit instruction
```

For small offsets there is a 16-bit instruction for bra.p. This can represent offsets from –256 to +254 in steps of 2.




### 8.9.1.3 Register

The new value of the program counter is taken from the indicated register.



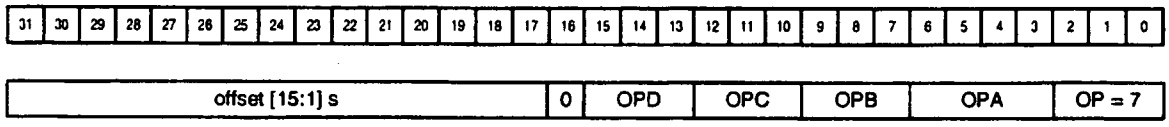

The address or offset in Ra should always be even. If not, the processor will generate an exception.

```
bra.r   Ra      // No flags set
        bra.r   %r4
        bsr.r   %r6
```

### *8.9.2 Conditional Branch Instructions*

All conditional branches are relative to the Program Counter (to the PC value of the following instruction). All instructions start on a 16-bit boundary. The compiler should only generate even branch offsets. The instruction can only represent even offsets. The signed offset allows functions to be up to 64kB.

There are 14 conditional branches. It is anticipated that the GCC compiler will only use the first 10 (in the opcode table below). The first 8 are available as 16-bit instructions with offset[8:1]s.




```
beq     label           //BinUtils will calculate offset[15:0] from this label
        beq     label782    // BinUtils will calculate offset[15:0] from label782
        blt.s   labelA      // BinUtils will calculate offset[15:0] from labelA
```

The conditional branches will occur when the flags are set as shown in the following tables :

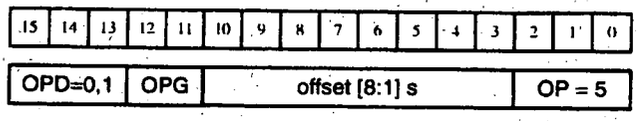
| *OPD* | *OPB* | *Mnemonic* | *Branch If ...* | *Flag Values Required for a Branch* |
|---|---|---|---|---|
| 0 | 4 | beq | Equal (Zero Set) | Z == 1 |
| 1 | 4 | bne | Not Equal (Zero Clear) | Z == 0 |
| 2 | 4 | ble.s | Less Than or Equal (signed) | (N != V) \|\| (Z == 1) |
| 3 | 4 | bgt.s | Greater Than (signed) | (N == V) && (Z == 0) |
| 4 | 4 | blt.s | Less Than (signed) | N != V |
| 5 | 4 | bge.s | Greater Than or Equal (signed) | N == V |
| 6 | 4 | ble.u | Less Than or Equal (unsigned) | (C == 1) \|\| (Z == 1) |
| 7 | 4 | bgt.u | Greater Than (unsigned) | (C == 0) && (Z == 0) |
| 0 | 5 | bcs (= blt.u) | Carry Set | C == 1 |
| 1 | 5 | bcc (= bge.u) | Carry Clear | C == 0 |
| 2 | 5 | bvs | Overflow Set | V == 1 |
| 3 | 5 | bvc | Overflow Clear | V == 0 |
| 4 | 5 | bmi | Minus (Negative Set) | N == 1 |
| 5 | 5 | bpl | Plus (Negative Clear) | N == 0 |

| *OPD* | *OPB* | *Mnemonic* | *Branch If ...* | *Flag Values Required for a Branch* |
|---|---|---|---|---|
| 6 | 5 | | | |
| 7 | 5 | | | |

For small offsets there are 16-bit instructions for 8 of the conditional branch instructions:

- beq, bne, ble.s, bgt.s, blt.s, bge.s, ble.u, bgt.u

These can represent offsets from –256 to +254 in steps of 2.



### 8.9.3 Load and Store Instructions

ld.i, ld.8z.i // Displacement Loads

ld.p, ld.8z.p // PC-Relative Loads

ld.r, ld.8z.r // Indexed Loads

st.i, st.8.i // Displacement Stores

st.p, st.8.p // PC-Relative Stores

st.r, st.8.r // Indexed Stores

The ld instructions allow data to be copied from memory (or memory mapped registers) to a specified register (Rd).

The st instructions allow data to be copied from a specified register (Rs) to memory.

For ld.8z.i, ld.8z.p and ld.8z.r, bits [15:8] of Rd are set to 0. The N flag is based on Rd[15] and will therefore always be set to 0.

All the ALU operations operate on complete 16-bit registers. The ld and st instructions support 16 and 8-bit operations. It is these 8-bit load instructions that enable the XAP4 to support char data types.

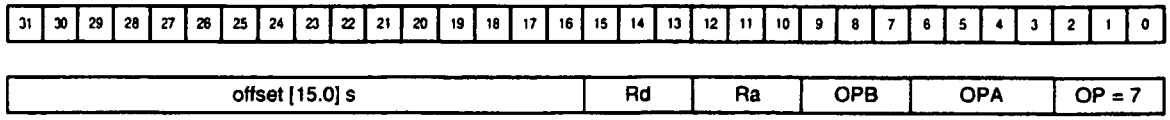
The only XAP4 instructions that can write to memory are:

- st.i, st.8.i, st.p, st.8.p, st.r, st.8.r
- stz.i, stz.8.i, stz.p, stz.8.p, stz.r, stz.8.r
- push
- swap.i
- blkcp.r, blkcps.r
- blkst.r, blkst.8.r

#### 8.9.3.1 Displacement Addressing (offset, Ra) – ld.*.i, st.*.i

Displacement Load and Store instructions are available for 16 and 8-bit data. 8-bit data is aligned to bit 0 (in register or memory).

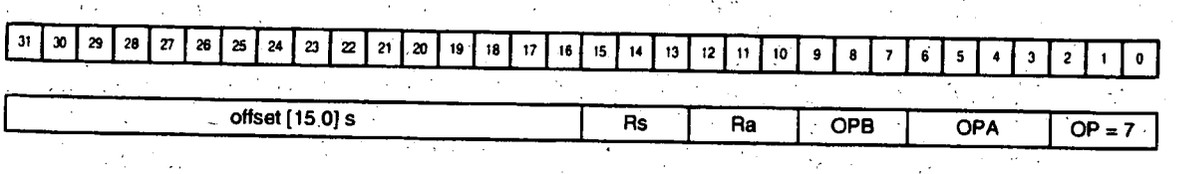
Offset and Ra are both interpreted as Byte addresses.




ld.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra)

ld.8z.i Rd, @(offset, Ra) // Load Rd from addr(offset + Ra), zero extend

```
ld.i        %r1, @(0x35d, %r7)
ld.8z.i     %r1, @(357, %r0)    // Forms the byte address (357 + R0).
```




```
st.i        Rs, @(offset, Ra)    // Store Rs to address (offset + Ra)
st.8.i      Rs, @(offset, Ra)    // Store Rs to address (offset + Ra)
    st.i        %r1, @(0x35d, %r7)
    st.8.i      %r1, @(357, %r0)    // Forms the byte address (357 + R0).
```

When using the 8-bit instructions (ld.8z.i, st.8.i) the requested size is indicated to the XAP4_mmu. Accesses to an 8-bit memory with ld.8z.i should result in only one memory access (rather than the two required for a 16-bit load).

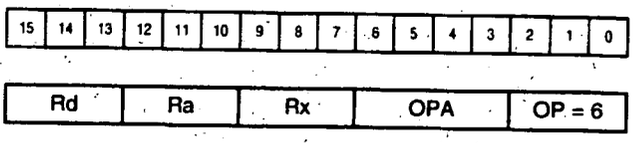
#### 8.9.3.2 Indexed Addressing (Rx, Ra) – ld.*.r, st.*.r

Indexed Load and Store instructions are available for 16 and 8-bit data. 8-bit data is aligned to bit 0 (in register or memory).

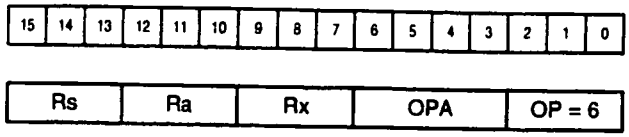
Ra is interpreted as a Byte address. Rx is scaled according to the width of the data being loaded or stored:

- Address = n*Rx + Ra
- ld.r, st.r  Rx is multiplied by 2
- ld.8z.r, st.8.r  Rx is not scaled

When using the 8-bit instructions (ld.8z.r, st.8.r) the requested size is indicated to the XAP4_mmu. Accesses to an 8-bit memory with ld.8z.r should result in only one memory access (rather than the two required for a 16-bit load).




```
ld.r        Rd, @(Rx, Ra)       // Load Rd from address (2*Rx + Ra)
ld.8z.r     Rd, @(Rx, Ra)       // Load Rd from address (Rx + Ra), zero extend
    ld.r        %r1, @(%r2, %r3)    // address = 2*R2 + R3
    ld.8z.r     %r1, @(%r0, %r5)    // address = R0 + R5
```




st.r Rs, @(Rx, Ra) // Store Rs to address (2*Rx + Ra)

st.8.r Rs, @(Rx, Ra) // Store Rs to address (Rx + Ra)

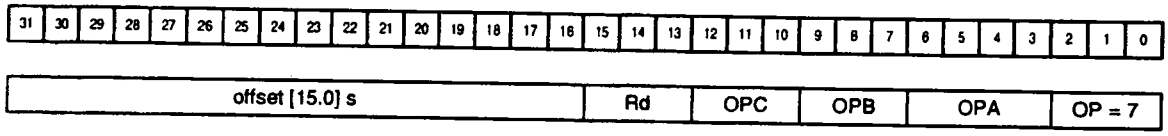
st.r %r1, @(%r2, %r3) // address = 2*R2 + R3

st.8.r %r1, @(%r0, %r5) // address = R0 + R5

### 8.9.3.3 PC-Relative Addressing (offset) – ld.*.p, st.*.p

PC-Relative Load and Store instructions are available for 16 and 8-bit data. 8-bit data is aligned to bit 0 (in register or memory).

Offset is added to PC and both are interpreted as Byte addresses.




ld.p Rd, @(offset) // Load Rd from addr(offset + PC)

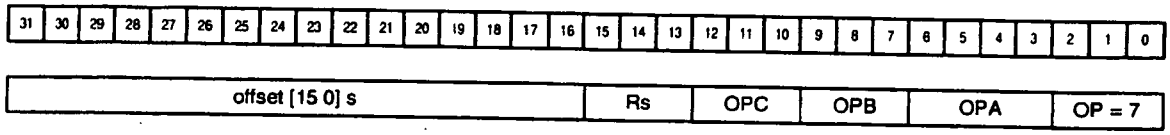
ld.p Rd, @(label) // Load Rd from addr(label)

ld.8z.p Rd, @(offset) // Load Rd from addr(offset + PC), zero extend

ld.8z.p Rd, @(label) // Load Rd from addr(label), zero extend

ld.p %r1, @(label) // Loads data at 'label' into R1.

ld.8z.p %r1, @(357) // Forms the byte address (357 + PC).




st.p Rs, @(offset) // Store Rs to address (offset + PC)

st.p Rs, @(label) // Store Rs to address (label)

st.8.p Rs, @(offset) // Store Rs to address (offset + PC)

st.8.p Rs, @(label) // Store Rs to address (label)

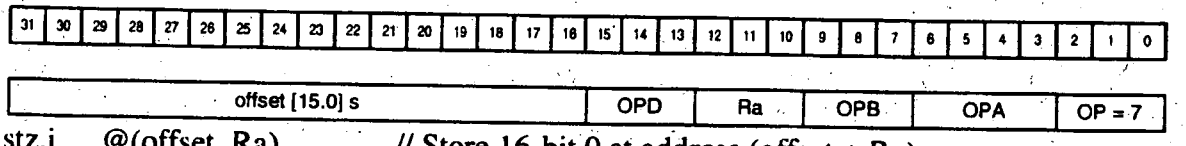
st.p %r1, @(label) // Stores R1 to 'label'.

```
st.8.p      %r1, @(357)  // Forms the byte address (357 + PC).
```

When using the 8-bit instructions (ld.8z.p, st.8.p) the requested size is indicated to the XAP4_mmu. Accesses to an 8-bit memory with ld.8z.p should result in only one memory access (rather than the two required for a 16-bit load).

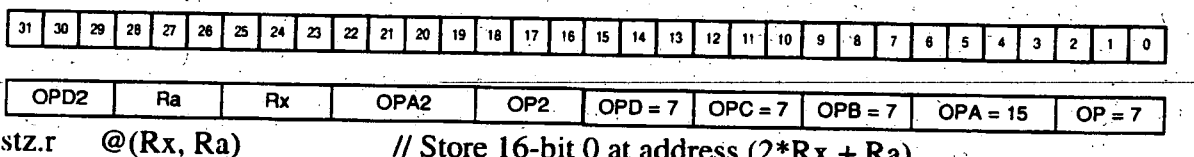
### 8.9.3.4 Store Zero Instructions

XAP4 has displacement and indexed forms of Store Zero Instructions. These directly store the value 0 to a specified address. The instructions operate the same way as the normal Store instructions, but without an Rs argument. There are instructions for 16-bit and 8-bit stores :

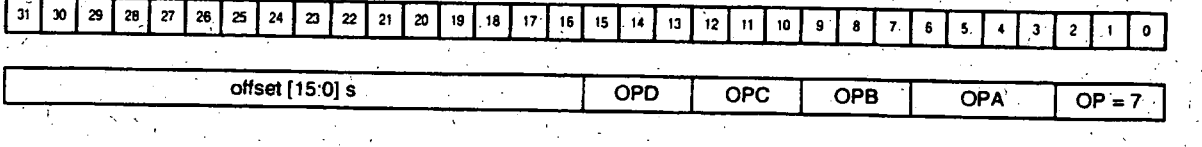



```
stz.i    @(offset, Ra)      // Store 16-bit 0 at address (offset + Ra)
stz.8.i  @(offset, Ra)      // Store 8-bit 0 at address (offset + Ra)
         stz.i @(36, %r3)   // Store 16-bit 0 at address (36 + R3)
         stz.8.i @(21, %r0) // Store 8-bit 0 at address (21 + R0)
```

There are 16-bit versions of the stz.*.i instructions available for Ra = R7.




```
stz.r    @(Rx, Ra)      // Store 16-bit 0 at address (2*Rx + Ra)
stz.8.r  @(Rx, Ra)      // Store 8-bit 0 at address (Rx + Ra)
         stz.r @(%r5, %r2)   // Store 16-bit 0 at address (2*R5 + R2)
         stz.8.r @(%r0, %r3) // Store 8-bit 0 at address (R0 + R3)
```




```
stz.p      @(offset)    // Store Rs to address (offset + PC)
stz.8.p    @(label)     // Store Rs to address (label)
     stz.p      @(label)    // Stores 0 to 'label'.
     stz.8.p    @(357)      // Forms the byte address (357 + PC).
```

### 8.9.3.5 swap.i Instruction

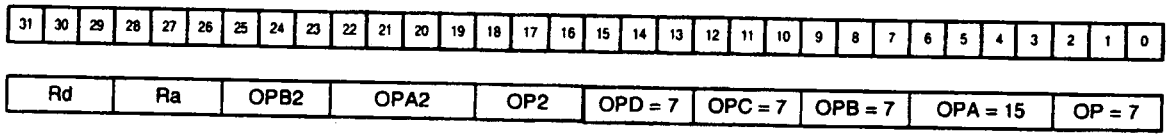
Performs an atomic swap between a register and a memory location.

This instruction asserts a signal from the Core to the xap4_mmu to request that the memory bus is not released. The atomicity of the instruction assumes a compliant xap4_mmu.

swap.i Rd, @(0, Ra)

The memory byte address is calculated as :

- Address = Ra // Data is 16-bit only.




This instruction is not intended to be used by the C compiler.

However, it is a very useful assembler instruction for implementing semaphores. This is because the instruction is atomic and therefore cannot be interrupted half way through execution.

### 8.9.4 push, pop, pop.ret Instructions

These instructions enable very compact code for stack operations and function entry and exit.

The Stack Pointer (SP) is %r7. The stack grows downwards in memory. i.e SP starts as a high address and becomes a lower address as the stack increases in size. This means that when accessing stack data, stack offsets will always be positive.

So the convention for the stack pointer is :

- Push data. Stack grows. Stack Pointer becomes more negative.
- Pop data. Stack shrinks. Stack Pointer becomes more positive.

The instructions have various modes. All are passed :

- A register list (the 7 registers R0-R6 available for push and pop).
- A stack pointer offset. This is a 4-bit unsigned number of 16-bit words. The instruction increases or decreases the stack pointer (%r7) by this offset in addition to 2n, for the n selected registers. So the stack pointer (byte address) can be changed by 0 to 30 by each of these instructions, in addition to the increase required for the selected registers.

In all these cases SP (%r7) is increased or decreased by (2n + offset), where n is the number of selected registers. Offset should be interpreted as 2*offset[4:1] (which is an unsigned number).

The push, pop and pop.ret instructions specify a fixed relationship between registers and memory :

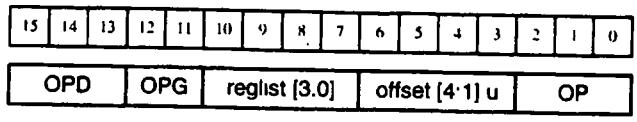
- Which registers are copied to which memory addresses by push.
- Which memory addresses are copied to which registers by pop and pop.ret.

However, they do not specify the order in which the copies should be done. The instructions are all atomic (non-interruptible) and so the copy order should be chosen to ease hardware design. Recommended copy orders are given in the following sections on the push, pop and pop.ret instructions.

The memory/register relationship used is :

- low registers mapped to low memory addresses (low stack offsets).
- high registers mapped to high memory addresses (high stack offsets)

### 8.9.4.1 push




reglist[3:0] represents the registers to be pushed, as follows :

| reglist[3:0] | Registers |
|---|---|
| 0 | {R0} |
| 1 | {R1} |
| 2 | {R2} |
| 3 | {R3} |
| 4 | {R4} |
| 5 | {R5} |
| 6 | {R6} |
| 7 | {} |
| 8 | {R3, R6} |
| 9 | {R3, R4} |
| 10 | {R3, R4, R6} |
| 11 | {R3, R4, R5} |
| 12 | {R3, R4, R5, R6} |
| 13 | {R1, R2} |
| 14 | {R0, R1, R2} |
| 15 | {R0, R1, R2, R3, R4, R5, R6} |

The register list (in the assembler) must be from low to high (R0 to R6).

The operation sequence is :

- Decrease stack pointer (%r7) by (offset + 2n), where n is the number of selected registers.
- Store selected registers to memory. %r6 first, to the highest address. %r0 last to the lowest address.

The mask is specified as a set of comma-separated registers (low to high) inside curly brackets. The mask cannot convey the order in which the registers will be pushed. The processor will push registers from %r6 first to %r0 last.

The assembler syntax for push.* is :

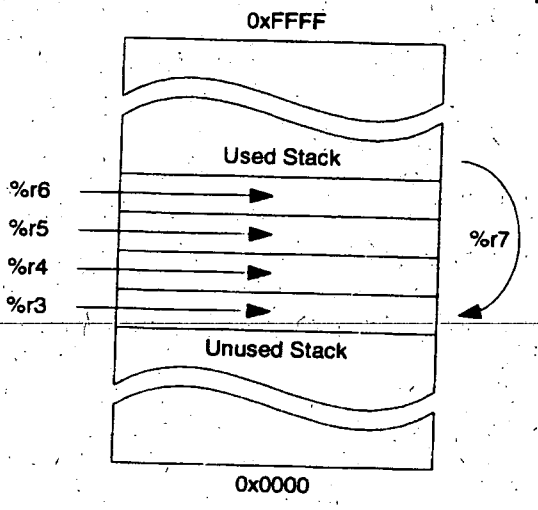
```
push    {register_list}, #offset[4:1]u // offset specified in bytes
```

#### 8.9.4.1.1 push example 1

```
push   {%r3-%r6}, #0        // Registers must be from low to high
```

This is equivalent to the following sequence of instructions.

```
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r6, @(0, %r7)      // Store R6 to stack memory
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r5, @(0, %r7)      // Store R5 to stack memory
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r4, @(0, %r7)      // Store R4 to stack memory
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r3, @(0, %r7)      // Store R3 to stack memory
```



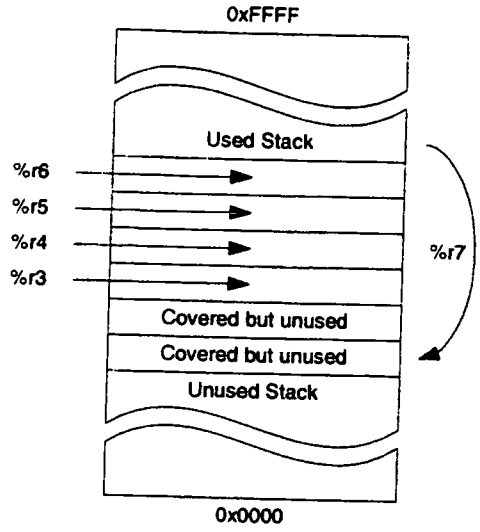

#### 8.9.4.1.2 push example 2

```
push   {%r3-%r6}, #4        // Registers must be from low to high
```

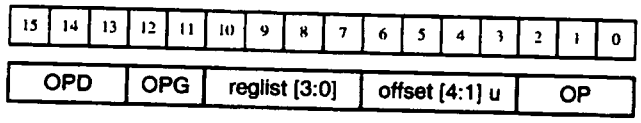
This is equivalent to the following sequence of instructions.

```
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r6, @(0, %r7)      // Store R6 to stack memory
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r5, @(0, %r7)      // Store R5 to stack memory
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r4, @(0, %r7)      // Store R4 to stack memory
add.i   %r7, %r7, #-2       // Decrease Stack Pointer for 1 register
st.i    %r3, @(0, %r7)      // Store R3 to stack memory
add.i   %r7, %r7, #-4       // Decrease Stack Pointer by #offset[4:1]u
```

// 4 bytes of stack free for other purposes




### 8.9.4.2 pop




reglist[3:0] represents the registers to be popped. The decode table is the same as that for the push instruction.

The register list (in the assembler) must be from low to high (R0 to R6).

The instruction is interpreted as follows :

| *Mnemonic* | *Meaning* |
|---|---|
| pop {reglist}, #offset | Load from stack memory to selected R0-R6. |

The operation sequence is :

- Load memory into selected registers. %r0 first from the lowest address. %r6 last, from the highest address.
- Increase stack pointer (%r7) by (offset + 2n), where n is the number of selected registers. This action may be distributed through the instruction.

The assembler syntax options for pop are :

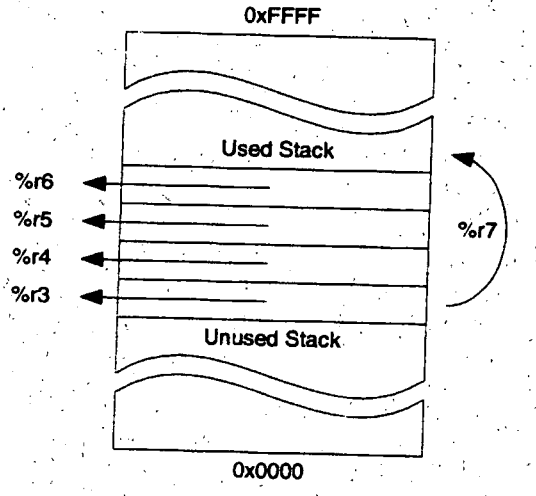
```
pop    {register_list}, #offset[4:1]u
```

#### 8.9.4.2.1 pop example 1

```
pop    {%r3-%r6}, #0        // Registers must be from low to high
```

This is equivalent to the following sequence of instructions :

```
ld.i    %r3, @(0, %r7)      // Load R3 from stack memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
ld.i    %r4, @(0, %r7)      // Load R4 from stack memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
ld.i    %r5, @(0, %r7)      // Load R5 from stack memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
ld.i    %r6, @(0, %r7)      // Load R6 from memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
```

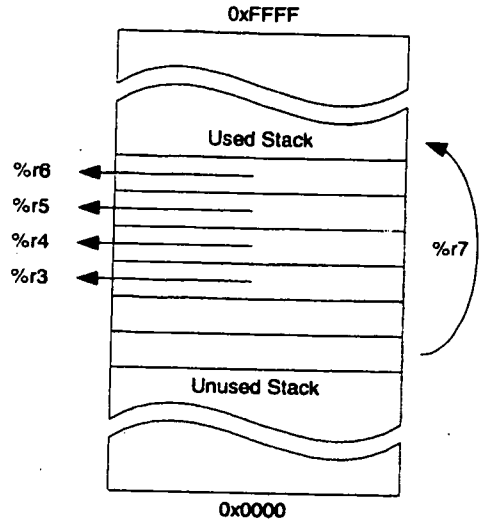



8.9.4.2.2 pop example 2

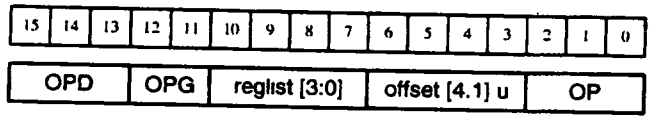
```
pop     {%r3-%r6}, #4       // Registers must be from low to high
```

This is equivalent to the following sequence of instructions :

```
add.i   %r7, %r7, #4        // Increase Stack Pointer by #offset[4:1]u
ld.i    %r3, @(0, %r7)      // Load R3 from stack memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
ld.i    %r4, @(0, %r7)      // Load R4 from stack memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
ld.i    %r5, @(0, %r7)      // Load R5 from stack memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
ld.i    %r6, @(0, %r7)      // Load R6 from memory
add.i   %r7, %r7, #2        // Increase Stack Pointer for 1 register
```




### 8.9.4.3 pop.ret




reglist[3:0] represents the registers to be popped. The decode table is the same as that for the push instruction.

The register list (in the assembler) must be from low to high (R0 to R6).

The instruction is interpreted as follows :

| *Mnemonic* | *Meaning* |
|---|---|
| pop.ret {reglist}, #offset | Load from stack memory to selected R0-R6.<br>Copy LR (R6) to PC. |

- Load memory into selected registers. %r0 first from the lowest address. %r6 last from the highest address.
- Increase stack pointer (%r7) by (offset + 2n), where n is the number of selected registers. This action may be distributed through the instruction.
- Copy %r6 (link register) into %pc (program counter). This is done even if %r6 is not in the register list. If %r6 is in the register list, then it is this popped value that is copied to %pc.

The assembler syntax options for pop are :

pop.ret {register_list}, #offset[4:1]u

#### 8.9.4.3.1 pop.ret example 1

```
pop.ret {%r3, %r5}, #0        // Registers must be from low to high
```

This is equivalent to the following sequence of instructions :

```
ld.i    %r3, @(0, %r7)        // Load R3 from stack memory
add.i   %r7, %r7, #2          // Increase Stack Pointer for 1 register
ld.i    %r5, @(0, %r7)        // Load R5 from stack memory
add.i   %r7, %r7, #2          // Increase Stack Pointer for 1 register
bra.r   %r6                   // Return to address in R6 (Link Register)
```

#### 8.9.4.3.2 pop.ret example 2

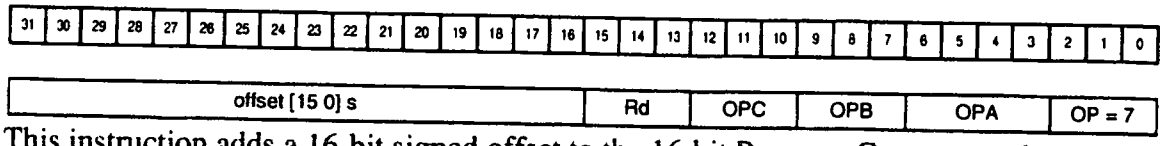
```
pop.ret {%r3, %r5}, #8        // Registers from low to high
```

This is equivalent to the following sequence of instructions :

```
add.i   %r7, %r7, #8          // Increase Stack Pointer by #offset[4:1]u
ld.i    %r3, @(0, %r7)        // Load R3 from stack memory
add.i   %r7, %r7, #2          // Increase Stack Pointer for 1 register
ld.i    %r5, @(0, %r7)        // Load R5 from stack memory
add.i   %r7, %r7, #2          // Increase Stack Pointer for 1 register
bra.r   %r6                   // Return to address in R6 (Link Register)
```

## 8.9.5 mov Instructions

### 8.9.5.1 mov.p – PC-relative Offset




This instruction adds a 16-bit signed offset to the 16-bit Program Counter (to the PC value of the following instruction). The 16-bit result is put into Rd.

mov.p is used to create function and data pointers. Functions will always start at even addresses, but data objects (chars) may start at any byte address. For this reason, mov.p can specify a full 16-bit offset. The hardware interprets all 16 bits. The signed offsets are from –32kB to +32kB.

The resultant address in the register is the absolute 16-bit address. This can be used as Ra in a ld or st instruction. It could also be used as a data or function pointer.

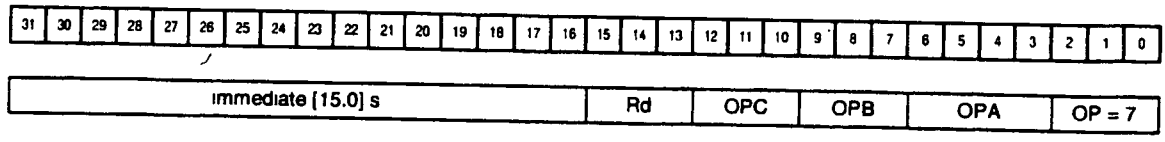
This instruction is very powerful as it can form any of these long addresses in a single cycle. The fact that it is PC-relative makes it suitable for generating Position Independent Code (PIC) or Relocatable Code.

This instruction does not update any flags.

```
mov.p  Rd, #label
mov.p  Rd, #offset[15:0]s      // Use the byte offset (bottom bit must be 0)
       mov.p  %r5, #label36    // Move (PC-relative label36) to R5
       mov.p  %r1, #0xe132     // Move (PC + 0xe132) to R1
```

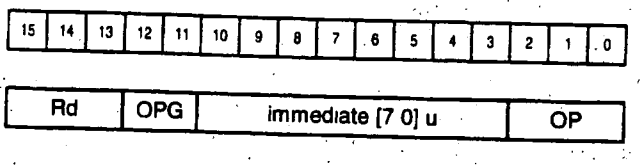
The offset in the instruction is the address difference between (the next instruction after the mov.p) and the label. After the mov.p has been executed, Rd will contain the absolute address of the label. This will work, regardless of what address the whole program has been loaded from, because it is PC-relative.

### 8.9.5.2 mov.i - Immediate

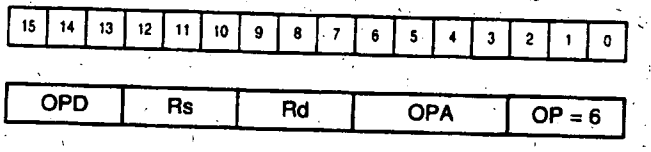



```
mov.i  Rd, #immediate[15:0]s   // Copy #immediate to Rd
       mov.i  %r4, #-356       // Load –356 into register R4
```

There is also a 16-bit instruction for mov.i with small immediate values.

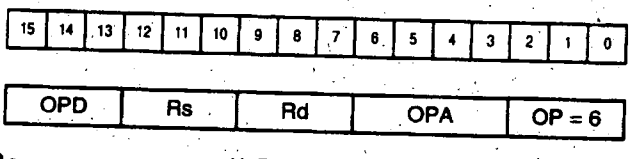



### 8.9.5.3 mov.r – Register




```
mov.r  Rd, Rs          // Copy Rs to Rd
       mov.r  %r3, %r7 // Copy register R7 to register R3
```

### 8.9.5.4 mov.32.r – Register




```
mov.32.r    Rd, Rs          // Copy {R(s+1), Rs} to {R(d+1), Rd}
       mov.32.r    %r3, %r0 // Copy register {R1, R0} to register {R4, R3}
       mov.32.r    %r3, %r4 // Copy register {R5, R4} to register {R4, R3}
```

Rd and Rs can be any value from R0-R5.

This instruction is executed in 1 clock cycle.

Note that the source and destination accumulators can overlap each other. i.e Rd can corrupt Rs.

### 8.9.6 ALU Instructions

The following arithmetic and logic instructions are provided. Each one specifies a destination register (Rd) in which to write back the result, and a source register (Rs) which provides one of the instruction operands. The second input operand may come from another register Rt (RRR format), or be specified as a 16-bit immediate value RRI format). These instructions all use the assembler format :

- op.r Rd, Rs, Rt // Rd = Rs op Rt
- op.i Rd, Rs, #imm // Rd = Rs op #imm

These must be written in full even when there is only one possible value for a particular argument.

#### 8.9.6.1 Logic Instructions

and.i, or.i, xor.i // 16 and 32-bit instructions

and.r, or.r, xor.r // 16-bit instructions

These instructions all use 16-bit input and output arguments. e.g :

- and.r %r1, %r6, %r2 // R1 = R6 & R2
- or.i %r7, %r3, #56 // R7 = R3 | 56
- xor.i %r4, %r2, #0xffa5 // R4 = R2 ^ 0xffa5

#### 8.9.6.2 Arithmetic Instructions

add.i, add.c.i, sub.x.i, sub.xc.i // 16 and 32-bit instructions

add.r, add.c.r, sub.r, sub.c.r // 16-bit instructions

These instructions all use 16-bit input and output arguments. For sub.x.i and sub.xc.i the order of the operands is eXchanged (reversed), enabling both (register – immediate with `add.i`) and (immediate – register with `sub.x.i`) to be performed. The flags are set as follows :

- Z (zero flag) set if the result is zero.
- N (negative flag) set if the result is negative (Rd[15] = 1).
- C (carry flag) set if the result of the unsigned operation is not correct.
- V (overflow flag) set if the result of the signed operation is not correct.

The carry versions of the instructions (add.c.i, add.c.r, sub.xc.i, sub.c.r) set the Z flag in a special way :

- If (result=0) and (Z=1), set Z=1. Otherwise set Z=0.

- This means that if (Z=0) before the instruction, then Z will always be 0 after the instruction (regardless of the instruction result).
- This is useful for 32-bit arithmetic.

Example instructions are :

- add.r %r1, %r6, %r2 // R1 = R6 + R2
- sub.x.i %r7, %r3, #56 // R7 = 56 - R3
- add.c.i %r4, %r2, #0xffa5 // R4 = R2 + 0xffa5 + C

#### 8.9.6.3 Multiply Instructions

mult.i, mult.32s.i, mult.32u.i // 32-bit instructions

mult.32s.r, mult.32u.r // 32-bit instructions

mult.r // 16-bit instruction

The input arguments (Rs, Rt, #imm) are always 16-bit.

mult.i and mult.r produce a 16-bit result in Rd. Rd can be specified to be any of the registers.

mult.32s.i and mult.32s.r produce a 32-bit signed result in any accumulator (register-pair). Rd, Rs, Rt and #imm are all treated as signed.

mult.32u.i and mult.32u.r produce a 32-bit unsigned result in any accumulator (register pair). Rd, Rs, Rt and #imm are all treated as unsigned.

All multiply instructions update the Z and N flags.

Example instructions are :

- mult.r %r1, %r6, %r2 // R1 = R6 * R2
- mult.32u.i %r1, %r3, #56 // {R2, R1} (unsigned) = R3 * 56
- mult.32s.r %r5, %r2, %r4 // {R6, R5} (signed) = R2 * R4

#### 8.9.6.4 Divide/Remainder Instructions

The divide and remainder instructions are all 32-bit instructions. There are signed and unsigned, .i (RRI) and .r (RRR) variants of the instructions to support 16/16 and 32/16 divides and remainders :

- div.* // Rd = 16-bit result of (Rs / Rt) or (Rs / #imm)
- rem.* // Rd = 16-bit remainder of (Rs / Rt) or (Rs / #imm)
- divrem.* // Rd = 16-bit result of (Rs / Rt) or (Rs / #imm)
  // R(d+1) = 16-bit remainder of (Rs / Rt) or (Rs / #imm)

The 24 instructions available are :

- div.s.i, div.u.i, div.s.r, div.u.r
- div.32s.i, div.32u.i, div.32s.r, div.32u.r // Numerator = 32-bit = {R(s+1), Rs}
- rem.s.i, rem.u.i, rem.s.r, rem.u.r
- rem.32s.i, rem.32u.i, rem.32s.r, rem.32u.r // Numerator = 32-bit = {R(s+1), Rs}
- divrem.s.i, divrem.u.i, divrem.s.r, divrem.u.r
- divrem.32s.i, divrem.32u.i, divrem.32s.r, divrem.32u.r
  // Numerator = 32-bit = {R(s+1), Rs}

The denominator (Rt or #imm) is always 16-bit. The numerator can be 16-bit or 32-bit. When the numerator is 16-bit, it can be any of the registers. When the numerator is 32-bit, it can be any accumulator (register-pair).

The result and remainder are always 16-bit. The result can be any register for the div.* instructions. The remainder can be any register for the rem.* instructions. The {remainder, result} can be any accumulator (register-pair) for the divrem.* instructions. For divrem.* the result is stored in Rd and the remainder is stored in R(d+1).

Example instructions are :

- div.u.r %r1, %r6, %r2 // R1 = R6 / R2 (unsigned)
- rem.32u.i %r1, %r3, #56 // R1 = {R4, R3} % 56 (unsigned)
- divrem.s.r %r1, %r2, %r4 // R1 = R2 / R4, R2 = R2 % R4 (signed)

The div.* and rem.* instructions do update the Z and N flags. The divrem.* instructions do not update the Z or N flags.

The unsigned instructions update the C flag. The signed instructions update the V flag. The C and V flags are set identically for the div.*, rem.* and divrem.* instructions.

When a divide (div.*, rem.*, divrem.*) by zero is attempted, the following will happen :

- Rd is set to 0.
- For divrem.*, R(d+1) is set to 0.
- For div.* and rem.*, Z flag is set to 1, N flag is set to 0.
- For signed divides, V flag is set to 1.
- For unsigned divides, C flag is set to 1.
- For User mode, the processor will throw a DivideByZero exception.

div.*, rem.* and divrem.* instructions are executed fortran-style :

- Make both operands unsigned

- Do the division
- Generate the correct sign for the answer

As defined in ISO 9899:1999 (E) C standard; all div and rem (signed and unsigned) work together such that :

- ((a/b)*b) + (a%b) = a

This is true for signed and unsigned versions of the instructions.

For signed divides (div.*s.*) the result and remainder truncate towards zero :

- Result (5 / 2) = 2, (-5 / 2) = -2, (5 / -2) = -2, (-5 / -2) = 2.
- Remainder (5 % 2) = 1, (-5 % 2) = -1, (5 % -2) = 1, (-5 % -2) = -1.
- The special div.s.* case of (0x8000 / -1) will be incorrect and should set Rd to 0 and set the overflow flag V to 1.
- The div.*s.* instructions will set the overflow flag V to 1 if the result is incorrect, otherwise to 0.

For unsigned divides (div.*u.*), the result and remainder also truncate towards zero :

- Result (5 / 2) = 2
- Remainder (5 % 2) = 1
- The div.*u.* instructions will set the carry flag C to 1 if the result is incorrect, otherwise to 0. If the result is incorrect, Rd is set to 0.

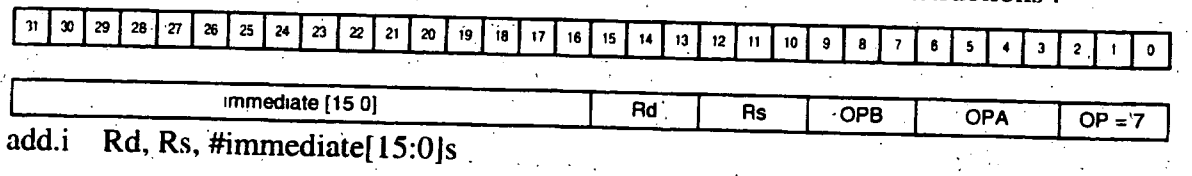
If the result of a 32/16 divide is too big to fit in the 16-bit Rd register, then the C or V flags will be set to indicate that the instruction was not completed correctly. The C flag is set for unsigned divides. The V flag is set for signed divides. In such cases, Rd should be set to 0.

#### 8.9.6.5 Reg-Reg-Immediate Format

Most immediate instructions use 16-bit immediates. The exceptions are cmp.8.* which use unextended 8-bit immediates :

The assembler code should always contain 16-bit immediates that are within range for the associated instruction. The compiler will generate assembler code using immediates that are within the correct range for the associated instruction.

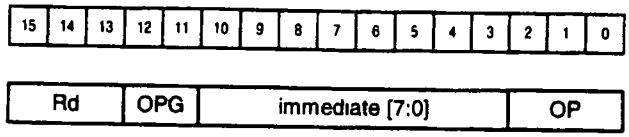
The instructions are formatted as follows for all ALU .i immediate instructions :




add.i Rd, Rs, #immediate[15:0]s

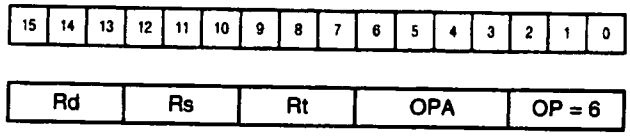
add.i %r5, %r7, #1260 // immediate will be sign-extended to 16-bit

Some instructions with short immediates are available in 16-bit instruction format :



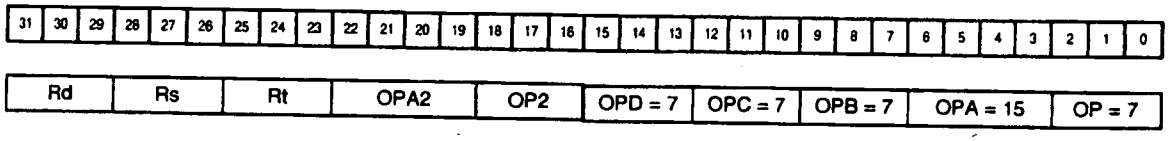

Some instructions interpret the immediate as signed, others interpret it as unsigned. This applies to both 16-bit and 32-bit RRI ALU instructions.

### 8.9.6.6 Reg-Reg-Reg Format




```
add.c.r Rd, Rs, Rt        // add.r and sub.r instructions are available in forms
                          // with and without Carry
xor.r   Rd, Rs, Rt        // Same format for and, or, xor logic instructions
        or.r    %r3, %r0, %r4     // R3 = R0 | R4
        mult.r  %r3, %r4, %r2     // R3 = R4 * R2
        xor.r   %r3, %r4, %r6     // R3 = R4 ^ R6
        and.r   %r5, %r4, %r7     // R5 = R4 & R7
```

Most RRR instructions use the above 16-bit format. However there are a number of RRR instructions that use the following 32-bit format. These incluse the ALU instructions mult.32*.r, div.*.r, rem.*.r and divrem.*.r :



### 8.9.7 Compare Instructions

The comparison instructions are used to set the processor's Flags. These are used by the conditional branch instructions. There are 16 and 8-bit compare instructions.

The 16-bit compare instructions (cmp.*.i, cmp.*.r) operate on all 16-bits

The 8-bit compare instructions (cmp.8*.i, cmp.8*.r) operate on the bottom 8-bits only.

Note that the order in which the two arguments are used in the comparison can be specified so that together the branch and comparison instructions allow all conditions to be tested for.

#### 8.9.7.1 Compare Assembler Format

There are 2 forms of the cmp instruction (immediate, register). These take parameters for OperandSize, CarryBit and Operand Order. These are interpreted as follows :

| *Operand Size* | *Assembler Parameter* | *Meaning* |
|---|---|---|
| 0 | 8 | 8-bit |
| 1 | 16 | 16-bit |

| *Carry Bit* | *Assembler Parameter* | *Meaning* |
|---|---|---|
| 0 | nocarry | Without Carry |
| 1 | carry | With Carry |

| *Operand Order* | *Assembler Parameter* | *Flags set on according to…* |
|---|---|---|
| 0 | normal | Rs – immediate[15:0]s |
| 1 | eXchange | immediate[15:0]s – Rs |

OperandOrder is only used for cmp.*.i immediate compare instructions. It is not necessary and therefore not used for the cmp.*.r register compare instructions (because the register order can be swapped).

Assembler mnemonics for the cmp instruction are of the form :

- cmp.*Parameters.Type*

The default parameters for the cmp instruction are :

- 16 bit
- No Carry
- Normal Order (not reversed)

Non-default parameters available are :

- 8 8-bit
- c With Carry

- x eXchange (Reverse) Order

Types specify the type of second ALU input. Types available (like all the major instructions) are :

- i Immediate
- r Register

So example cmp instructions are :

- cmp.r Register, 16-bit, No Carry, Normal Order
- cmp.c.i Immediate, 16-bit, With Carry, Normal Order
- cmp.8xc.i Immediate, 8-bit, With Carry, eXchange Order

#### 8.9.7.2 Compare Flags

The compare instructions update the Z, N, C and V flags based on the compare result, using the normal rules as defined in section 5.7.

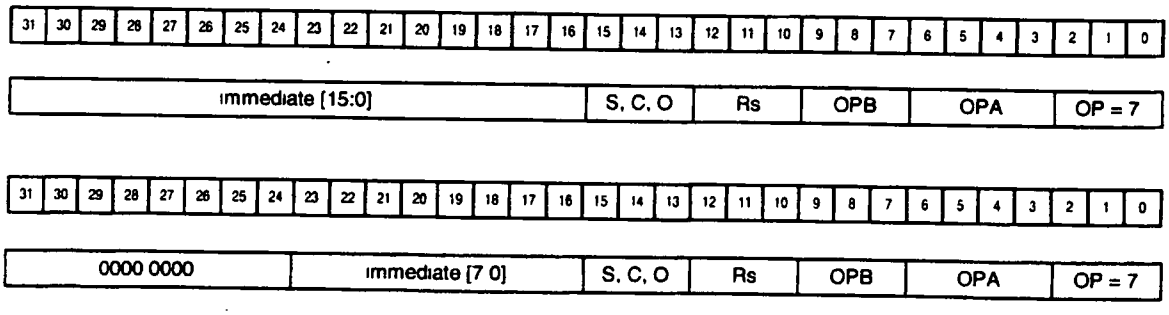
All of the 'compare with carry' instructions (cmp.*c.*) use the following rule to update the Z flag :

- If (result=0) and (Z=1), set Z=1. Otherwise set Z=0.
- This means that if (Z=0) before the instruction, then Z will always be 0 after the instruction (regardless of the instruction result).
- This is useful for 32-bit arithmetic.

#### 8.9.7.3 Compare Immediate

There are 2 formats for the compare immediate instructions :

| *Format* | *Instructions* | *Compare Rs with ...* | *Extend Immediate ?* |
|---|---|---|---|
| 16-bit immediate | cmp.i, cmp.c.i, cmp.x.i, cmp.xc.i | imm[15:0] | No |
| 8-bit low immediate | cmp.8.i, cmp.8c.i, cmp.8x.i, cmp.8xc.i | imm[7:0] | No |



The XAP4 C-compiler supports 32-bit variables and constants. The XAP4 will support a comparison between a 32-bit variable (e.g stored in {R2, R1}) and a 32-bit constant (e.g 0x12345678) as follows :

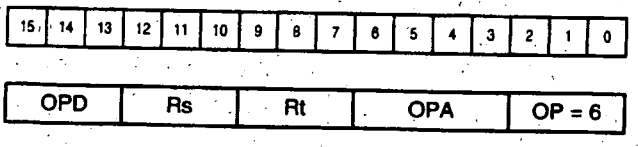
```
cmp.i     %r1, #0x5678
cmp.c.i   %r2, #0x1234
```

After this, the flags will have been set correctly for the 32-bit comparison.

cmp.*.i Rs, #immediate[15:0]s

```
cmp.c.i   %r3, #0xAbc3     // 16-bit compare, With Carry
```

#### 8.9.7.4 Compare Register




Flags are always set on Rs – Rt.

8-bit register compares only compare the bottom 8 bits of Rs and Rt.

cmp.*.r Rs, Rt

```
cmp.c.r   %r3, %r7     // 16-bit compare with Carry of (R3 – R7)
```

### 8.9.8 Shift and Rotate Instructions

The XAP4 has signed (arithmetic) and unsigned (logical) shift right instructions. It also has shift left and rotate left instructions. There is one set of instructions for 16-bit registers and another set for 32-bit accumulators.

The flags should be updated as follows for the shift and rotate instructions :

- 16-bit shifts/rotates – Rs = 16-bit, Rd = 16-bit, msb = bit 15.
- 32-bit shifts/rotates – Rs = 32-bit, Rd = 32-bit, msb = bit 31.

For left and right shifts, the N flag is updated for the msb of Rd.

For left and right shifts, the Z flag is updated for the whole of Rd.

For left shifts, the last bit shifted out of the msb of Rs will be copied to the Carry flag. If the register is shifted by 0, the Carry Flag will not be modified.

For right shifts, the C flag equals the bit last shifted out of bit 0 of Rs.

The rotatel instruction does not circulate through the carry bit. It sets the flags as follows :

- C = final value of bit 0 of Rd.
- N = final value of msb of Rd.
- Z = 1 if the contents of Rd = 0, otherwise Z = 0.

#### 8.9.8.1 16-bit Shift and Rotate Instructions

shiftl.i, shiftr.s.i, shiftr.u.i, rotatel.i

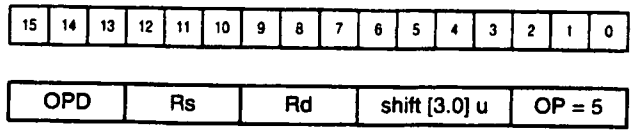
shiftl.r, shiftr.s.r, shiftr.u.r, rotatel.r

The 16-bit shift instructions can shift by 0 to 15 bits. The .i instructions specify this shift amount as an unsigned immediate. The .r instructions take the unsigned shift amount from Rt[3:0].

The input register (Rs) can be R0-R6.

The output register (Rd) can be R0-R6.

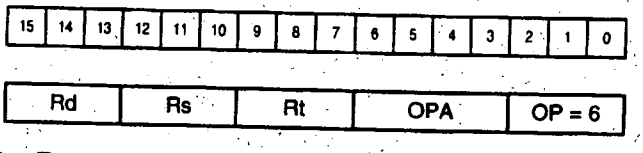
##### 8.9.8.1.1 Reg-Reg-Immediate




```
shiftl.i    Rd, Rs, #shift      // Rd = Rs << #shift[3:0]
shiftl.i    %r3, %r2, #shift    // Shift R2 to left by #shift, store result in R3
```

```
shiftr.s.i %r3, %r2, #shift    // Sign Shift R2 to right by #shift, store result in R3
shiftr.u.i %r3, %r2, #shift    // Unsign Shift R2 to right by #shift
                               // Store result in R3
```

#### 8.9.8.1.2 Reg-Reg-Reg




```
shiftl.r    Rd, Rs, Rt          // Rd = Rs << Rt[3:0]
    shiftl.r   %r3, %r3, %r5    // Shift R3 to left by value in R5, store result in R3
    shiftr.s.r %r3, %r2, %r5    // Sign Shift R2 to right by R5, store result in R3
    rotatel.r  %r3, %r2, %r5    // Rotate R2 left by value in R5, store result in R3
```

### 8.9.8.2 32-bit Shift and Rotate Instructions

shiftl.32.i, shiftr.32s.i, shiftr.32u.i, rotatel.32.i

shiftl.32.r, shiftr.32s.r, shiftr.32u.r, rotatel.32.r

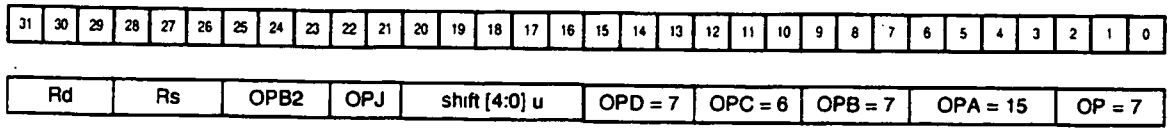
The 32-bit shift instructions can shift by 0 to 31 bits. The .i instructions specify this shift amount as an unsigned immediate. The .r instructions take the unsigned shift amount from Rt[4:0].

These instructions operate on any 32-bit accumulator (register-pair).

The input register (Rs) can be R0-R5.

The output register (Rd) can be R0-R5.

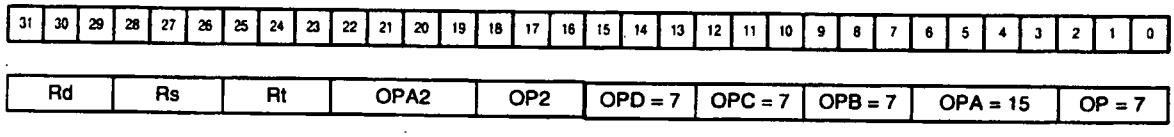
#### 8.9.8.2.1 Reg-Reg-Immediate




shiftl.32.i Rd, Rs, #shift // {R(d+1), Rd} = {R(s+1), Rs} << #shift[4:0]

```
shiftl.32.i %r0, %r1, #shift    // Shift {R2, R1} to left by #shift
                                // Store result in {R1, R0}
shiftr.32s.i %r2, %r1, #shift   // Sign Shift {R2, R1} to right by #shift
                                // Store result in {R3, R2}
rotatel.32.i  %r5, %r3, #shift  // Rotate {R4, R3} left by #shift
                                // Store result in {R6, R5}
```

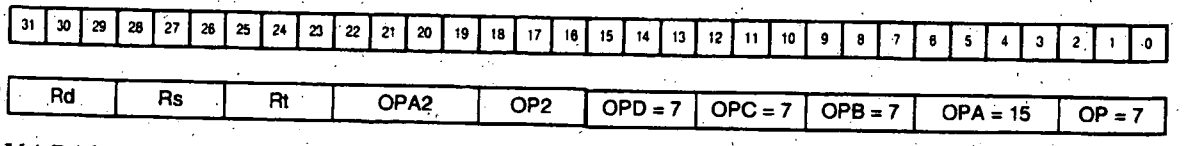
#### 8.9.8.2.2 Reg-Reg-Reg




shiftl.32.r Rd, Rs, Rt // {R(d+1), Rd} = {R(s+1), Rs} << Rt[4:0]

```
shiftl.32.r %r3, %r3, %r5     // Shift {R4, R3} to left by value in R5,
                              // store result in {R4, R3}
shiftr.32u.r %r5, %r2, %r4    // Unsign Shift {R3, R2} to right by R4,
                              // store result in {R6, R5}
```

### 8.9.9 Block Instructions




XAP4 has a number of interruptable block instructions. These enable very fast and low power consumption memory copy and memory initialisation. The block instructions all have the same format as shown above.

When these functions are interrupted, they complete the current copy or store and then move to the interrupt routine. After the interrupt routine, the program returns to the block instruction and continues executing it until complete. The instruction behaviour for given Rd, Rs, Rt values is identical for a re-start or start of execution. The instructions do not have a concept of a 'middle' state. They simply consist of repeated copies or stores. Each individual copy or store cannot be interrupted.

The block instructions do not update any flags.

#### 8.9.9.1 blkcp.r – Block Copy

```
blkcp.r  Rd, Rs, Rt     // Copies Rt bytes from memory starting at Rs
                        // to memory starting at Rd
```

This instruction is useful for implementing the C library function memcpy().

Rd, Rs and Rt are all treated as 16-bit unsigned numbers.

- Rd is the destination pointer in bytes.
- Rs is the source pointer in bytes.
- Rt is the number of bytes to be copied.

The instruction copies 16-bits at a time if it can, otherwise 8-bits :

- If Rd and Rs are both even it uses 16-bit copies from the start.
- If Rd and Rs are both odd, it starts with an 8-bit copy and then continues with 16-bit copies.
- If Rd is odd and Rs even (or vice-versa), all copies will be 8-bit.
- After each copy, Rd and Rs are incremented, Rt is decremented. The increment/decrement amount is 2 for 16-bit copies, 1 for 8-bit copies.
- The copy process is repeated until Rt is 0. When Rt is 1, an 8-bit copy is done.
- At the end of the instruction, Rd and Rs have been incremented by Rt from their initial values.

- It is valid for the instruction to be executed with Rt = 0. In this case nothing will happen (no memory copied or registers updated). The program will simply proceed to the next instruction.

When an interrupt occurs, the instruction completes the current copy and register updates.

When most instructions are interrupted, the address of the next instruction is copied to R6_I. However, for the block instructions, the address of the current instruction is copied to R6_I. All other interrupt procedures are the same as for all other instructions.

When the rtie instruction returns from the interrupt routine, the blkcp.r instruction will continue executing, repeating its copy loop until Rt is 0. There is no difference between re-starting an interrupted blkcp.r instruction and starting a blkcp.r instruction with the same values of Rd, Rs and Rt.

#### 8.9.9.2 blkcps.r – Block Copy String

```
blkcps.r  Rd, Rs, Rt    // Copies from memory starting at Rs to memory starting at Rd
                        // until a 0 is copied or until Rt bytes have been copied.
```

This instruction is useful for implementing the C library functions strcpy() and strncpy().

The blkcps.r instruction is the same as blkcp.r except that :

- It will stop copying when it encounters any byte equal to 0 in the source stream (0 is the standard string-terminator). The encountered 0 does get copied, but no subsequent bytes.

If it does not encounter a 0 in the source stream, it will continue until Rt = 0 (same as blkcp.r).

This means that the final values of Rd, Rs will have been incremented by the number of bytes copied. Rt will have been decremented by the number of bytes copied.

#### 8.9.9.3 blkst.r - Block Store

```
blkst.r  Rd, Rs, Rt     // Stores Rs[15:0] multiple times to memory starting at Rd.
                        // Stores Rt bytes in total.
```

Rd and Rt are treated as 16-bit unsigned numbers.

- Rd is the destination pointer in bytes.
- Rs is the source register.
- Rt is the number of bytes to be stored.

The instruction stores Rs multiple times to memory starting at Rd. This is normally used for memory initialisation. Memory will normally be initialised to 0, so Rs will normally be set to 0 before executing blkst.r.

memset() expects to initialise all the specified memory range to the same byte value. When blkst.r is used for this purpose, the high and low bytes of Rs should have the same value. However, it may be easier to implement memset() with blkst.8.r.

The instruction stores 16-bits at a time if it can, otherwise 8-bits :

- If Rd is even it does 16-bit stores from the start.
- If Rd is odd, it starts with an 8-bit store and then continues with 16-bit stores.
- After each store, Rd is incremented, Rt is decremented. The increment/decrement amount is 2 for 16-bit stores, 1 for 8-bit stores.
- The store process is repeated until Rt is 0. When Rt is 1, an 8-bit store is done.
- At the end of the instruction, Rd has been incremented by Rt from its initial value.
- It is valid for the instruction to be executed with Rt = 0. In this case nothing will happen (no stores or registers updated). The program will simply proceed to the next instruction.
- Rs[15:0] is stored repeatedly to memory starting at the Rd address.
- If Rd is even, Rs[7:0] is stored to even addresses, Rs[15:8] is stored to odd addresses.
- If Rd is odd, Rs[7:0] is stored to odd addresses, Rs[15:8] is stored to even addresses.
- The instruction may change the Rs value.

When an interrupt occurs, the instruction completes the current store and register updates.

When most instructions are interrupted, the address of the next instruction is copied to R6_I. However, for the block instructions, the address of the current instruction is copied to R6_I. All other interrupt procedures are the same as for all other instructions.

When the rtie instruction returns from the interrupt routine, the blkst.r instruction will continue executing, repeating its store loop until Rt is 0. There is no difference between re-starting an interrupted blkst.r instruction and starting a blkst.r instruction with the same values of Rd, Rs and Rt.

#### 8.9.9.4 blkst.8.r – Block Store (8-bit)

blkst.8.r Rd, Rs, Rt // Stores Rs[7:0] multiple times to memory starting at Rd.
// Stores Rt bytes in total.

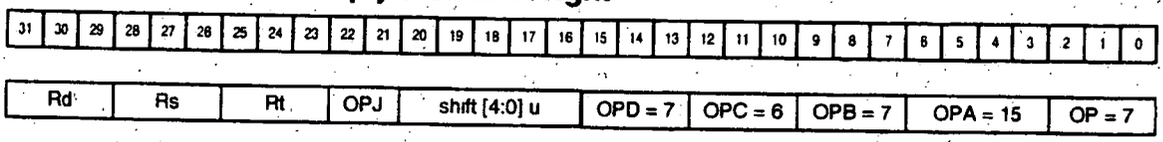
This instruction is useful for implementing the C library function memset().

The blkst.8.r instruction is the same as blkst.r except that :

- It stores Rs[7:0] to all specified memory addresses.
- It never stores Rs[15:8] to memory.

## *8.9.10 DSP Instructions*

### 8.9.10.1 mult.sh.r – Multiply then shift right



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | OPJ | | shift [4:0] u | | | OPD = 7 | | | OPC = 6 | | | OPB = 7 | | | OPA = 15 | | | | | OP = 7 | | |

mult.sh.r Rd, Rs, Rt, #shift // Rd = (Rs * Rt) >> #shift

mult.sh.r does the following :

- 16 * 16 signed multiply of Rs and Rt into a temporary 32-bit register.
- Signed shift right the 32-bit register by the #shift parameter (0 to 31).
- Writes the bottom 16 bits to Rd.
- Updates Z and N flags based on the value of Rd.
- Updates V flag for overflow. This should be set to 1 after the shift, if Rd[31:16] is not equal to Rd[15]. Otherwise it should be set to 0.

This is useful when multiplying coefficients and data, then normalising to pick out the resultant 16 bits of interest. This can then be added to a 32-bit accumulator by using add.r followed by add.c.r.

### *8.9.11 Special Register Instructions*

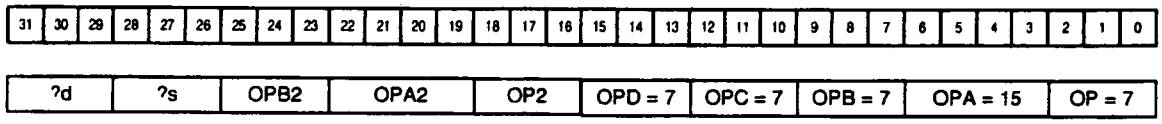
The movr2s and movs2r instructions allow the privileged modes to access special registers that are not part of the normal register model. This includes the flags.

Similarly, the movb2r and movr2b instructions allow the privileged modes to access the breakpoint registers.

| *Instruction* | *Operation* |
|---|---|
| movr2s | Move Normal Register to Special Register |
| movs2r | Move Special Register to Normal Register |
| movr2b | Move Normal Register to Breakpoint Register |
| movb2r | Move Breakpoint Register to Normal Register |

The coding and assembler syntax is given for the Special Registers in section 5.6 and for the Breakpoint registers in section 5.5.

Executing these instructions in User Mode will cause an IllegalInstruction exception.




movr2s Sd, Rs

movs2r Rd, Ss

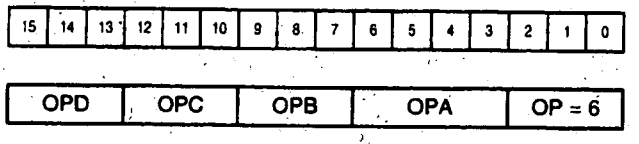
movr2b Bd, Rs

movb2r Rd, Bs

```
movs2r   %r1, %flags      // Move FLAGS to R1
movr2s   %sflags_s, %r2   // Move R2 to SFLAGS_S
movb2r   %r1, %brk0       // Move BRK0 to R1
movr2b   %brk0, %r2       // Move R2 to BRK0
```

## 8.9.12 System Instructions

### 8.9.12.1 16-bit System Instructions

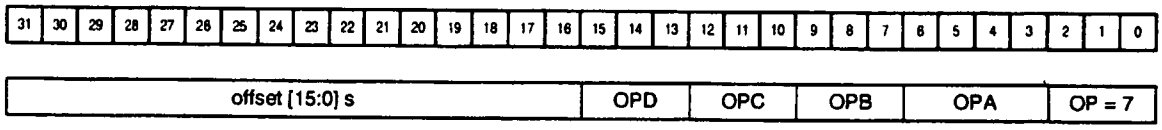



| *Instruction* | *Meaning* |
|---|---|
| rtie | Return from interrupt or exception. The current mode's link register is written to the PC and SFLAGS register is written to the FLAGS register. This instruction is not permitted in User Mode. |
| brk | If the processor is in User Mode and the B bit of flags is set, the break exception is thrown. Otherwise, if the processor RUN_STATE is RunToBreak the processor is stopped. |
| halt | Stops the processor always. This instruction is not permitted in User Mode. |
| sleepnop | Puts the processor into sleep mode. The WAKE_UP signal restarts the processor. During the sleep mode no SIF accesses to memory or registers are possible. This instruction is not permitted in User Mode. |
| sleepsif | Puts the processor into sleep mode. The WAKE_UP signal restarts the processor. SIF accesses are possible during the sleep mode. This instruction is not permitted in User Mode. |
| nop | No operation. |
| sif | NOP instruction left clear for SIF read or SIF write. This instruction is not permitted in User Mode. |
| irqe | Interrupt Enable. Sets the E bit of the FLAGS register to 1. This instruction is not permitted in User Mode. |
| irqd | Interrupt Disable. Clears the E bit of the FLAGS register to 0. This instruction is not permitted in User Mode. |
| irqs Rd | Interrupt Status. |
| print.r Rs | xIDE simulator prints value in Rs (interpreted as a char). Hardware treats instruction as NOP. |
| ver Rd | XAP4 Version number written to Rd. The returned 16-bit value should be interpreted as follows :<br>• Rd[15:12] = Hardware type. XAP4a = 0, XAP4b = 1.<br>• Rd[11:8] = Major Release Number, 0-15.<br>• Rd[7:0]= Minor Release Number, 0-255. |
| lic Rd | XAP4 Licence number written to Rd. This number is different for each customer chip design. |

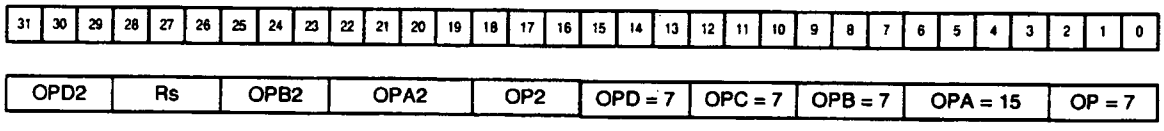
### 8.9.12.2 32-bit System Instructions

The 2 trap instructions pass a number (in a register or as an immediate), that is intended to be used by an operating system to indicate a particular service or function. The processor copies this number to R0_S. Additional arguments for the function can be passed in the User mode registers R1-R5 (this is just a software convention).

Further details on the trap instructions can be found in section 6.2.




| *Instruction* | *Meaning* |
|---|---|
| trap.i #imm | Cause the trap exception. Allows User Mode to call Supervisor Mode functions. This is executed as a nop in Supervisor and Interrupt Modes. |




| *Instruction* | *Meaning* |
|---|---|
| trap.r Rs | Cause the trap exception. Allows User Mode to call Supervisor Mode functions. This is executed as a nop in Supervisor and Interrupt Modes. |

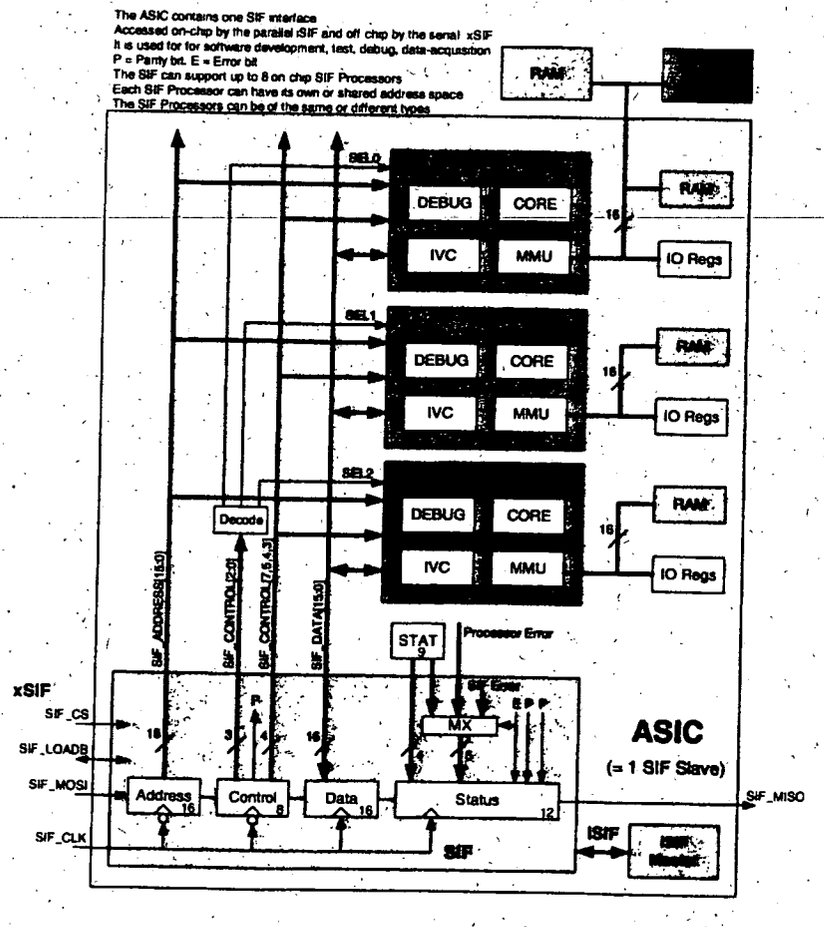
# 9 SIF Serial InterFace

A XAP4 ASIC is a single SIF Slave as defined in C4862-TM-005. The ASIC only contains one SIF interface (with 4 or 5 external pins). The default SIF provided with XAP4 can support up to 8 on-chip SIF Processors. These can all be XAP4 or can be a variety of different processors. Each can have its own or shared address space. The SIF can be accessed by the external serial xSIF interface or by the internal parallel iSIF interface.

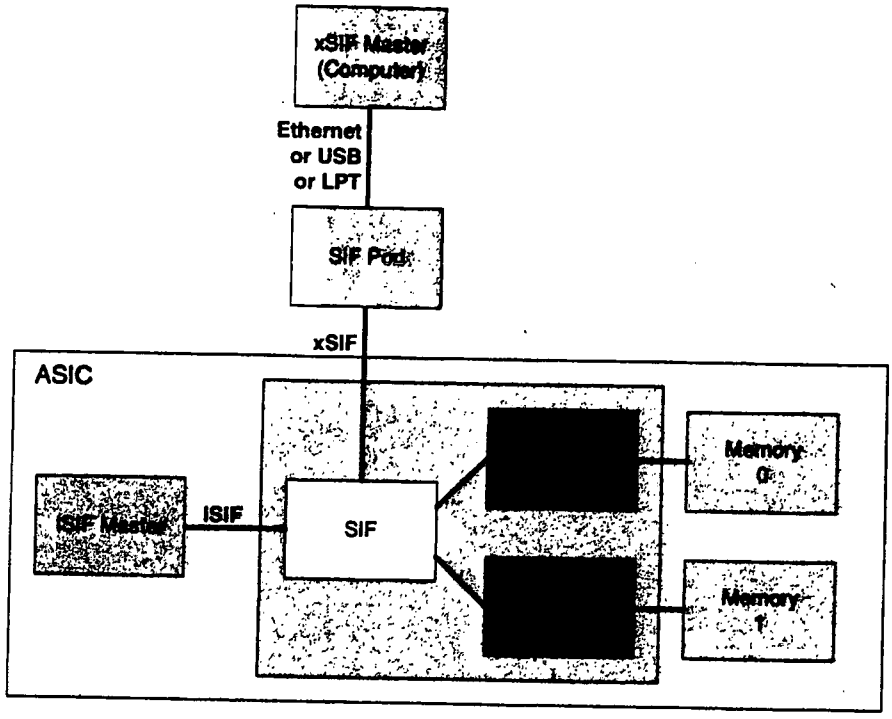
The SIF allows the user to debug software and to read or write the registers, memory and IO registers of all the on-chip SIF Processors. The SIF is also used for data acquisition and analogue production test of the ASIC. All memory and IO register reads and writes are performed in a non-invasive manner. The processors continue to operate in functional mode without having their timing affected.

## 9.1 SIF ASIC Block Diagram

Here is a block diagram of an ASIC that contains two XAP4 and one XAP5 processor, each with their own address space. It shows how the single SIF interface is used to access all 3 processors. The SIF_CS pin is not needed for a Single Slave SIF system.



## 9.2 xSIF and iSIF usage modes




The SIF may be used in the following usage modes :

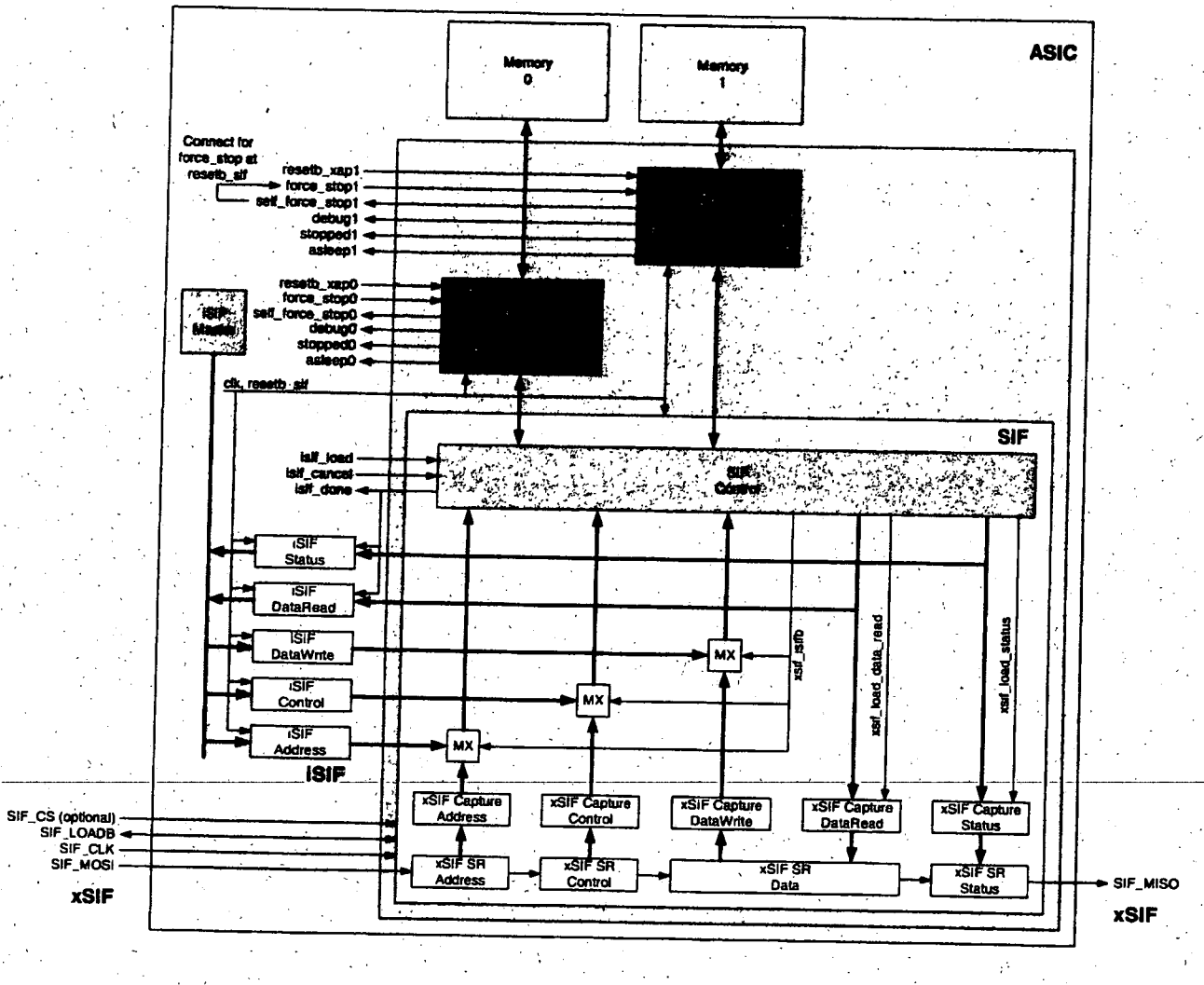
| *No* | *xSIF* | | *iSIF* | | *Comment* |
|---|---|---|---|---|---|
| | Debug | Normal | Debug | Normal | |
| 1 | Yes | Yes | No | No | iSIF not connected |
| 2 | Yes | Yes | No | Yes | iSIF used for memory access only. e.g for program download or data acquisistion. |
| 3 | No | No | Yes | Yes | xSIF not connected |
| 4 | No | Yes | Yes | Yes | xSIF used for memory access only. e.g for program download or data acquisition. |

Either xSIF or iSIF may be used for Debug SIF instructions, but not both. The most common modes will be 1 or 2.

Parity checking is performed on xSIF but not on iSIF. iSIF does not check PARITY_AC. It does generate PARITY_D and PARITY_S, but the iSIF Master may ignore them if it wishes.

xSIF may cancel stuck xSIF or iSIF instructions. iSIF may cancel stuck iSIF instructions, but not stuck xSIF instructions.

In all other respects, xSIF and iSIF behave identically. They access equivalent SIF fields (Address, Control, Data, Status) and use the same SIF instruction codes.




## 9.3 SIF ASIC Modes

At any time, the ASIC has :

- One SIF Mode (because the ASIC only contains one SIF).
- A SIF Processor Mode for each Processor.

### *9.3.1 SIF Modes*

There are 2 SIF Modes; Normal and Command. These modes are properties of the SIF and are independent of the processor types or states. More information on these modes can be found in C4862-TM-005. For the purpose of readability, the 3 statements in each row are defined to be equivalent :

| *Full Statement* | *Computer Style* | *Shortened Statement* |
|---|---|---|
| The SIF is in Normal SIF Mode | SIF Mode = Normal | The SIF is in Normal mode |
| The SIF is in Command SIF Mode | SIF Mode = Command | The SIF is in Command mode |

#### 9.3.1.1 SIF Mode = Normal

The ASIC resets with SIF in Normal mode. The properties of Normal SIF Mode are :

- SIF Operation is called a SIF Instruction.
- The SIF shift register length can be different from one ASIC design to another. This allows the Address, Control, Data and Status fields to have different lengths for different ASIC designs. The default shift register lengths are; XAP1 = 36 bits, XAP2 = 64 bits, XAP3 = 88 bits, XAP4 = 52 bits, XAP5 = 68. The rest of this document assumes that a 52-bit SIF shift register is being used.
- SIF can access all the on-chip processors and their associated memory spaces.
- SIF can execute Direct SIF Instructions. These are executed within the SIF itself and do not access a processor. Direct SIF Instructions require the DEBUG bit to be 1 and the PROCESSOR bits to be 0.

#### 9.3.1.2 SIF Mode = Command

Command mode is available to xSIF but not to iSIF. The xSIF Master can issue a special sequence of 256 bits on SIF_MOSI to put the SIF into Command mode. The properties of Command SIF Mode are :

- SIF Operation is called a SIF Command.
- It uses an 8-bit shift register. This length is the same for all SIF devices.
- Commands are 8 bits in to SIF_MOSI.
- Responses are 8 bits out from SIF_MISO.
- It is used to query or configure the SIF.
- It cannot access the processors or their associated memory spaces.
- It is used to reveal the hardware details of a particular SIF Slave (field lengths for Address, Control, Data, Status, Polarity of Write bit, etc).
- It is used by the xSIF Master (e.g xIDE) at the beginning of a SIF communication. This allows the xSIF Master to automatically discover the SIF devices without needing any manual configuration.

- Once the xSIF Master has discovered all that it needs to know about the SIF Slave, it will put the SIF back into Normal mode. It will normally remain in Normal mode for the rest of the xSIF communication.

### *9.3.2 XAP4 SIF Processor Modes*

Most processors have 2 SIF Processor Modes; Normal and Debug. At any time each processor has its own SIF Processor Mode. Each processor's mode can only be changed by the SIF interface (xSIF or iSIF).

The SIF Processor can be a variety of different processors. This specification describes the behaviour of the XAP4 in Normal and Debug SIF Processor Modes. For the purpose of readability, the 3 statements in each row are defined to be equivalent :

| *Full Statement* | *Computer Style* | *Shortened Statement* |
|---|---|---|
| The XAP4 is in Normal SIF Processor Mode | XAP4 SIF Processor Mode = Normal | The XAP4 is in Normal mode |
| The XAP4 is in Debug SIF Processor Mode | XAP4 SIF Processor Mode = Debug | The XAP4 is in Debug mode |

#### 9.3.2.1 XAP4 SIF Processor Mode = Normal

The ASIC resets with all XAP4 processors in Normal mode. This allows the SIF Masters (xSIF and iSIF) to read and write address-mapped variables (normally IO registers or memory) of the selected XAP4. These can be 8 or 16-bit.

If the XAP4 is in Normal mode, then it will be running. It can only be stopped when in Debug mode.

#### 9.3.2.2 XAP4 SIF Processor Mode = Debug

The selected XAP4 is put into Debug mode by executing the 'Debug Enable' SIF Instruction (issued by xSIF or iSIF Master). It will then remain in Debug mode until it executes the 'Debug Disable' SIF Instruction (when it will return to Normal mode).

Some SIF Instructions can only be executed if the XAP4 is already in Debug mode. In practise, Debug mode is only used during software development and ASIC test. Debug mode allows the user to perform functions such as start, stop, set breakpoint, run to breakpoint, read and write XAP4 internal registers and flags, etc.

It is important to distinguish between (Normal/Debug SIF Instructions) and (Normal/Debug XAP4 SIF Processor Modes) :

- A Normal SIF Instruction is when the DEBUG bit (bit 19 of SIF shift register) = 0. This is for 8 and 16-bit reads and writes to any memory address.

- A Debug SIF Instruction is when the DEBUG bit (bit 19 of SIF shift register) = 1. This is for all SIF instructions apart from the reads and writes to memory.
- Only certain SIF Instructions (Normal Instructions + a few Debug Instructions) can be executed when the XAP4 is in Normal mode. The XAP4 is in Normal mode after reset and after the 'Debug Diasable' SIF Instruction.
- All SIF Instructions can be executed when the XAP4 is in Debug Mode. The XAP4 is in Debug Mode after the 'Debug Enable' SIF Instruction.

## 9.4 When are SIF Operations executed ?

### *9.4.1 SIF Mode = Command*

Command mode is available to xSIF but not to iSIF. When the SIF is in Command Mode, SIF Operations (= SIF Commands) are executed immediately. As soon as the 8-bit Command has been shifted in to SIF_MOSI, the 8-bit Response will shift out on SIF_MISO. There is never a wait period. SIF_LOADB is not used. This is why SIF Commands cannot access processors or memory space. They are normally only used to query the SIF configuration (i.e to read a small ROM inside the SIF).

### *9.4.2 SIF Mode = Normal*

When the xSIF is in Normal mode, the SIF Operation (= SIF Instruction) consists of 3 stages as follows :

1. **Shift In Stage**. SIF Master shifts SIF Instruction (Address/Control for Read, Address/Control/Data for Write) into SIF Slave with SIF_CLK and SIF_MOSI.
2. **Handshake Stage**. SIF Master pulls SIF_LOADB low for a short pulse. SIF Slave holds SIF_LOADB low until the SIF Instruction has been completed. The SIF passes the SIF Instruction on to the selected SIF Processor (e.g XAP4 identified by CONTROL[2:0]). When/if the selected processor completes the SIF Instruction, it tells the SIF( Slave) which lets go of SIF_LOADB. SIF_LOADB goes high, telling the SIF Master that the SIF Instruction has been completed. Sometimes the handshake is not completed, which is called a SIF Lockup.
3. **Shift Out Stage**. SIF Master shifts the result (Data/Status for Read, Status for Write) out of SIF Slave with SIF_CLK and SIF_MISO.

In the second stage (Handshake) the xSIF Slave holds SIF_LOADB low for varying amounts of time. This depends upon the type of SIF Instruction and whether the selected XAP4 is stopped or running. As soon as the selected XAP4 executes the SIF Instruction, it tells the xSIF (Slave). The xSIF can then let go of SIF_LOADB, allowing it to return high again. The same handshake process is used for Direct SIF Instructions (which do not access any processor).

The sequence for iSIF is similar to that just described for xSIF :

1. **Parallel Write Stage**. iSIF Master writes to Address, DataWrite and Control iSOF Registers.
2. **Handshake Stage**. Same as for xSIF.
3. **Parallel Read Stage**. iSIF Master reads from DataRead and Status iSIF Registers.

#### 9.4.2.1 Immediate SIF Handshake

In some cases the selected XAP4 executes the SIF Instruction immediately :

- All SIF Instructions if the selected XAP4 is currently executing a 'sleepsif' instruction (in Normal or Debug Mode, Stopped or Running). This implies that the XAP4 is waiting for a 'wake_up' hardware input (to move the processor on to its next instruction).
- All SIF Instructions if the selected XAP4 is stopped (which can only happen in Debug Mode).
- Debug SIF Instructions where (Address = 0x0 to 0xFF). This includes Debug Enable, Debug Disable, Start, Stop etc. (see XAP4 SIF Instruction table for details). Such instructions are executed immediately even if the selected XAP4 is running.
- Direct SIF Instructions.

The following SIF Instructions are error conditions which are recognised by the hardware (selected processor or SIF itself). The hardware will respond immediately, returning the appropriate Processor or SIF error code :

- Parity error in PARITY_AC
- Undefined SIF Instruction (Combination of Control and Address fields).
- Wrong data-value for SIF Instructions that require special data-values.
- SIF Write to a read-only register.
- SIF Read from a write-only register.
- Debug SIF Instruction which requires the selected XAP4 to be in Debug mode, when it is actually in Normal mode.
- Debug SIF Instruction which requires the selected XAP4 to be stopped, when it is actually running.

#### 9.4.2.2 Delayed SIF Handshake

In most cases the selected XAP4 waits for a 'sif' or 'sleepsif' XAP4 instruction before executing the SIF Instruction :

- Normal SIF Instructions when the XAP4 is running (Normal or Debug mode).

#### 9.4.2.3 Incomplete SIF Handshake (ending in SIF Lockup)

The following SIF Instructions are error conditions which cannot be recognised by the hardware (selected processor or SIF itself). In such cases the SIF Slave will permanently hold SIF_LOADB low :

- Normal SIF Instruction when the selected XAP4 is running but never executes a 'sif' or 'sleepsif' instruction.

In such cases the selected XAP4 will never tell the SIF that it has completed the SIF Instruction, so the SIF Slave will never release SIF_LOADB, which the SIF Master will see (and will know that this is being done by the selected SIF Slave). This has the effect

of locking up the SIF Interface. The SIF Master must clear this with a SIF Cancel before it can proceed with any further SIF Instructions.

#### 9.4.2.4 SIF Cancel (to clear SIF Lockup)

The SIF has locked up because the SIF Master issued an invalid SIF Instruction to the selected SIF Processor in the selected SIF Slave.

The xSIF Master must be able to detect and recover from such a situation by :

- Having a timeout if SIF_LOADB stays low for too long.
- Issueing a SIF Cancel to clear the lockup in the SIF Slave (or Slaves)

There are 2 types of xSIF Cancel (both described in more detail in C4862-TM-005) :

- SIF Master toggles SIF_CLK 32 times (this will clear all Slaves that are locked up). SIF_MOSI[7:0] indicates a Cancel to xSIF or iSIF or both.
- SIF Master pulls SIF_CS low (not implemented in SIF Slaves before 2005). This will only cancel xSIF.

In both cases, the SIF Slave will release SIF_LOADB, allowing it to go high again. At this point the SIF lockup has been cleared. This allows the SIF Master to issue further SIF Instructions.

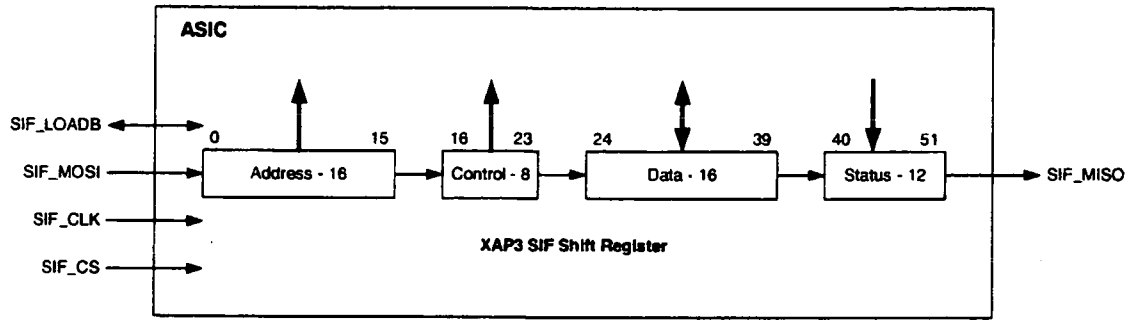
The iSIF must also be able to detect and recover from a lockup situation by :

- Having a timeout if isif_done stays low for too long.
- Pulling isif_cancel high to clear the lockup in the SIF Slave (or Slaves)

It is possible for the requested SIF Instruction (debug or normal mode, read or write) to be completed even after the Master has started a SIF Cancel. This means that when a Master issues a SIF Cancel, it cannot know whether the previously requested SIF Instruction will happen or not. The Master can assume that if the previous SIF instruction was executed by the Slave, then it will be the correct one requested.

## 9.5 SIF Shift Register

XAP4 ASICs contain an xSIF shift register which consists of 4 fields, in total 52-bits long:




iSIF has the equivalent fields. The field sizes and bit allocations are the same as for the xSIF shift register. However, iSIF splits the Data field into 2 separate fields for DataWrite and DataRead.

### *9.5.1 Address Field*

The 16-bit Address field is interpreted as follows for different SIF Instructions :

- Normal SIF Instructions – Address to read or write in units of the selected processor (e.g bytes or 16-bit words).
- Debug SIF Instructions – OpCode for a particular operation

### *9.5.2 Control Field*

The 8-bit Control field, CONTROL[7:0] consists of :

- {WRITE, PARITY_AC, SIZE[1:0], DEBUG, PROCESSOR[2:0]}.

All SIF transactions are one of :

- SIF Read
- SIF Write

The WRITE bit is 1 for SIF Write Instructions and 0 for SIF Read Instructions. For all modes, the only way that the SIF can update any part of the ASIC is with a SIF Write. SIF Reads cannot update any part of the ASIC.

The PARITY_AC bit is set by the xSIF Master and checked by the xSIF Slave. If the Slave finds a parity error it should set ERROR, ERROR_TYPE and ERROR_CODE[3:0] in the Status field. iSIF ignores PARITY_AC, so the iSIF Master does not need to set it.

The 2 SIZE bits indicate whether the data is 8 or 16-bit. The SIZE bits are interpreted for all Normal SIF Instructions (i.e reads and writes to memory and IO registers), whether the selected XAP4 is in Normal or Debug mode. Debug SIF Instructions which

have a Data parameter must set the SIZE bits correctly. Debug SIF Instructions which do not have a Data parameter must set the SIZE bits to 0.

The DEBUG bit is 1 for Debug SIF Instructions and 0 for Normal SIF Instructions.

The 3 PROCESSOR bits select which of the 8 SIF Processors the SIF Instruction should be passed to. All SIF Instructions must select a SIF Processor. Only One SIF Processor can be accessed at a time. There is no state. Successive SIF Instructions can access any SIF Processors in any order. SIF Processor Addresses should be allocated from 0 upwards. If the ASIC only contains one XAP4, it should be at SIF Processor Address 0.

### 9.5.3 Data Field

The 16-bit Data field is used for SIF Reads and Writes (from or to the specified Memory Address or Register).

For security reasons, some Debug SIF Instructions require the Data field to be set to a specific value.

The Slave should only update the Data field after valid SIF Reads. If there has been any kind of error (i.e ERROR = 1), the Data field should not be updated.

### 9.5.4 Status Field

The 12-Bit Status field, STATUS[11:0] format depends upon whether there has been an error in the SIF Instruction or not. The Slave SIF has a 9-bit input from the main part of the ASIC called STAT[8:0].

The format of the Status field is independent of the specified Address and Control fields. Its format depends upon STAT[8:0] and on whether there was a SIF Error.

Under the various conditions, the SIF should format the Status field as follows :

| *STATUS Bits* | *STATUS Description* | *No Error* | *SIF Error* | *Processor Error* |
|---|---|---|---|---|
| 3:0 | STATUS[3:0] | STAT[3:0] | STAT[3:0] | STAT[3:0] |
| 7:4 | ERROR_CODE[3:0] | STAT[7:4] | SIF Error Code | Proc Error Code |
| 8 | ERROR_TYPE | STAT[8] | 0 | 1 |
| 9 | ERROR | 0 | 1 | 1 |
| 10 | PARITY_D | Data Parity | Data Parity | Data Parity |
| 11 | PARITY_S | Status Parity | Status Parity | Status Parity |

PARITY_D is set first, based on DATA[15:0].

PARITY_S is then set based on STATUS[10:0].

The Slave should update the Status field after all completed (SIF_LOADB handshake) SIF Instructions. This should happen whether there is an error or not.

### *9.5.5 Shift Register Bit Definitions*

The XAP4 xSIF shift register is defined as follows. All bits reset to 0 unless specified otherwise :

| *Bit* | *Name* | *Comments* |
|---|---|---|
| 0 | ADDRESS[0] | Memory address for Normal SIF Instructions. Address is in units of selected SIF Processor (i.e bytes for XAP4 or XAP3, 16-bit words for XAP2 or XAP1).<br><br>Parameter for Debug SIF Instructions (see SIF Instruction Table). |
| ... | | |
| 15 | ADDRESS[15] | |
| 16 | PROCESSOR[0] | PROCESSOR[2:0] = CONTROL[2:0].<br>Selects which SIF Processor (out of 8) to use.<br>SIF Processor addresses should start from 0 and work up to 7. |
| 17 | PROCESSOR[1] | |
| 18 | PROCESSOR[2] | |
| 19 | DEBUG | DEBUG = CONTROL[3].<br>0 = Normal SIF Instruction<br>1 = Debug SIF Instruction |
| 20 | SIZE[0] | SIZE[1:0] = CONTROL[5:4].<br>Determines the data length of the SIF access.<br>00 = 8-bit, 01 = 16-bit,10 = reserved, 11 = reserved |
| 21 | SIZE[1] | |
| 22 | PARITY_AC | PARITY_AC = CONTROL[6].<br>Resets to 1.<br>Set by xMaster to be odd parity (odd number of 1s) with:<br>• CONTROL[7]<br>• CONTROL[5:0]<br>• ADDRESS[15:0]<br>iSIF Master should always set this bit to 0. |

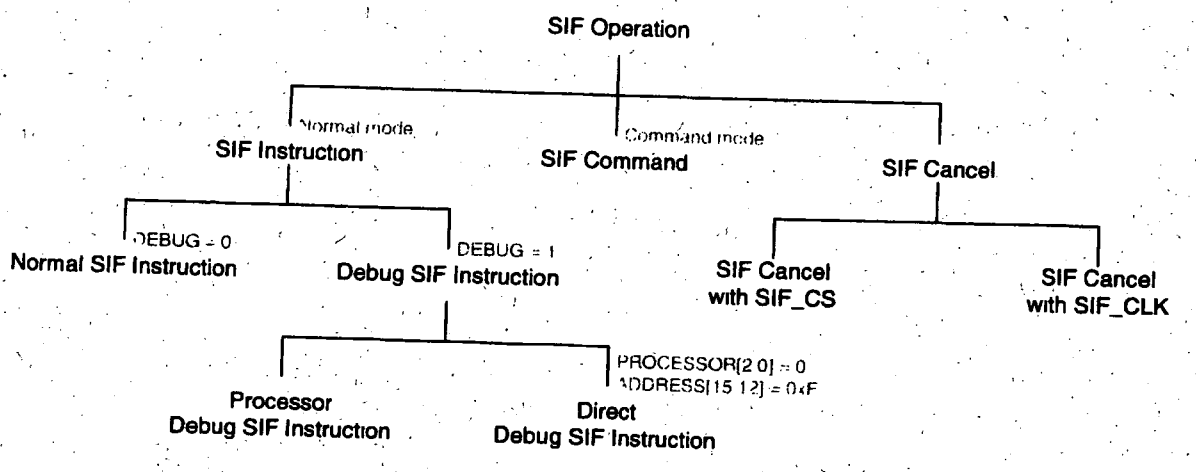
| | | |
|---|---|---|
| 23 | WRITE | WRITE = CONTROL[7].<br>0 = Read, 1 = Write<br>*Note that this is the opposite polarity from XAP1 and XAP2. The SIF_MOSI pad is tied low, so in XAP4 a random SIF transaction will cause a read and not a write.* |
| 24<br>...<br>39 | DATA[0]<br><br>DATA[15] | 16-bit: DATA[15:0] is written to or read from the selected address.<br>8-bit Write: DATA[7:0] is written to selected address.<br>8-bit Read: DATA[15:8] = 0, DATA[7:0] = data read from selected address. |
| 40<br>...<br>43 | STAT[0]<br><br>STAT[3] | STAT[3:0] = STATUS[3:0]<br>After all completed SIF Instructions, STATUS[3:0] is updated with the hardware status word STAT[3:0], independent of the selected Address and Control fields.<br>This will often be used for a TimeStamp, where STAT[0] is the highest frequency bit. |
| 44<br>45<br>46<br>47 | ERROR_CODE[0]<br>ERROR_CODE[1]<br>ERROR_CODE[2]<br>ERROR_CODE[3] | ERROR_CODE[3:0] = STATUS[7:4] is updated with :<br>• STAT[7:4], if there is no error.<br>• SIF Error Code, if there is a SIF error.<br>• Processor Error Code, if there is a Processor error. |
| 48 | ERROR_TYPE | ERROR_TYPE = STATUS[8] is updated with :<br>• STAT[8], if there is no error.<br>• 0, if there is a SIF error.<br>• 1, if there is a Processor error. |
| 49 | ERROR | ERROR = STATUS[9].<br>• 0 = No error. SIF Instruction completed successfully.<br>• 1 = Error. SIF or selected Processor detected an error in the requested SIF Instruction. |
| 50 | PARITY_D | PARITY_D = STATUS[10].<br>Resets to 1.<br>Set by Slave to be odd parity (odd number of 1s) with:<br>• DATA[15:0]<br>This is set before PARITY_S. |

| | | |
|---|---|---|
| 51 | PARITY_S | PARITY_S = STATUS[11].<br>Set by Slave to be odd parity (odd number of 1s) with:<br>• STATUS[10:0]<br>This is set after PARITY_D. |

## 9.6 SIF Instructions

### *9.6.1 SIF Instruction Types*

SIF Operations can be categorised as follows :




XAP4 SIF instructions are defined by the Control and Address shift register fields.

There are 2 types of SIF Instruction, defined as follows :

- Normal SIF Instructions DEBUG = 0
- Debug SIF Instructions DEBUG = 1

A particular kind of Debug SIF Instruction is the Direct SIF Instruction. These are executed in the SIF itself and do not access any processor. Direct SIF Instructions all format the shift register bits as follows :

- DEBUG = 1
- PROCESSOR[2:0] = 0
- ADDRESS [15:12] = 0xF

All other Debug Instructions refer to the selected processor. Such Debug SIF Instructions can be split into 3 categories :

- Address < 0x80 : No mode requirements.
- 0x7F < Address < 0x100 Selected XAP4 must be in Debug mode.
- 0xFF < Address < 0x1000 Selected XAP4 must be in Debug mode and stopped.

Like the XAP4 itself, SIF reads and writes can use any byte address for all word sizes. The words do not need to be memory-aligned. i.e :

- Addresses for 16-bit words do not need to be a multiple of 2

### 9.6.2 SIF Instruction Table

The following XAP4 SIF Instructions (Normal then Direct then other Debug) cover a subset of possible values for the Control and Address fields. The table does not include the PARITY_AC or WRITE bits in the Control field. The table assumes that PROCESSOR[2:0] = 0. All unspecified values are reserved for future use. Instructions marked * are only implemented when the SIF is connected to more than one processor.

| *Control [5:0]* | *Address* | *Data* | *Required XAP4 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x00 | 0xXXXX | 8-bit | - | RW | Byte Memory Read/Write |
| 0x10 | 0xXXXX | 16-bit | - | RW | 16-bit Memory Read/Write |
| 0x18 | 0xF010 | 0xD5IF | - | W | Reset Everything.<br>Pulses all rst_xap, rst_usergates outputs high for one clk.<br>Pulses rst_sif, rst_offchip outputs high for one clk.<br>See below for required connections and behaviour for rst_xap and rst_sif. |
| 0x18 | 0xF012 | 0xD5IF | - | W | Reset Off-Chip Devices.<br>Pulses rst_offchip output high for one clk. |
| 0x18 | 0xF013 | 0xD5IF | - | W | Reset ASIC.<br>Pulses all rst_xap, rst_usergates outputs high for one clk.<br>Pulses rst_sif output high for one clk.<br>See below for required connections and behaviour for rst_xap and rst_sif. |

| *Control [5:0]* | *Address* | *Data* | *Required XAP4 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x18 | 0xF015 | 0xD5IF | - | W | Reset SIF.<br>Pulses rst_sif output high for one clk. This should be connected to resetb_sif which will :<br>• Set RUN_STATE[1:0] = 0 = Running.<br>• Leaves MODE[1:0] unchanged. |
| 0x18 | 0xF081 | 0xD5IF | - | W | * Stop all processors |
| 0x18 | 0xF082 | 0xD5IF | - | W | * Run all processors |
| 0x18 | 0xF083 | 0xD5IF | - | W | * Wakeup all processors |
| 0x18 | 0xF084 | 0xD5IF | - | W | * Run to Break all processors |
| 0x18 | 0xF085 | 0xD5IF | - | W | * Set BreakMode = 0 = Stop one processor |
| 0x18 | 0xF086 | 0xD5IF | - | W | * Set BreakMode = 1 = Stop all processors |
| 0x18 | 0xF087 | 16-bit | - | R | * Read BreakMode |
| 0x18 | 0x0 | 0xD0FF | - | W | Debug Disable<br>• Puts XAP4 into Normal mode<br>• Sets XAP4 Running |
| 0x18 | 0x1 | 0xDBE | - | W | Debug Enable<br>• Puts XAP4 into Debug mode |
| 0x18 | 0x2 | 16-bit | - | RW | Status Register |
| 0x18 | 0x3 | 16-bit | - | R | Version Number |
| 0x18 | 0x4 | 16-bit | - | R | Licence Number |
| 0x18 | 0x5 | 16-bit | - | R | Register PC |
| 0x18 | 0x6 | 16-bit | - | R | Last PC<br>• Address of last completed instruction |
| 0x18 | 0x7 | 16-bit | - | RW | self_force_stop output.<br>Any write sets output low.<br>Read indicates current value. |
| 0x08 | 0x81 | - | Debug | W | Stop |

| *Control [5:0]* | *Address* | *Data* | *Required XAP4 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x08 | 0x82 | - | Debug | W | Run |
| 0x08 | 0x83 | - | Debug | W | Wakeup |
| 0x08 | 0x84 | - | Debug | W | Single Step |
| 0x08 | 0x85 | - | Debug | W | Run to Break |
| 0x08 | 0x91 | - | Debug | W | Reset selected XAP. Pulses selected rst_xap output high for one clk. This should be connected to resetb_xap which will :<br>• Leave RUN_STATE[1:0] unchanged.<br>• Set MODE[1:0] = 0 = Supervisor mode. |
| 0x08 | 0x92 | - | Debug | W | Reset selected User Gates. Pulses selected rst_usergates output high for one clk. |
| 0x18 | 0x100 | 16-bit | - | R | Register R0 current mode |
| 0x18 | 0x100 | 16-bit | Debug Stopped | W | Register R0 current mode |
| 0x18 | 0x101 | 16-bit | - | R | Register R1 |
| 0x18 | 0x101 | 16-bit | Debug Stopped | W | Register R1 |
| 0x18 | 0x102 | 16-bit | - | R | Register R2 |
| 0x18 | 0x102 | 16-bit | Debug Stopped | W | Register R2 |
| 0x18 | 0x103 | 16-bit | - | R | Register R3 |
| 0x18 | 0x103 | 16-bit | Debug Stopped | W | Register R3 |
| 0x18 | 0x104 | 16-bit | - | R | Register R4 |
| 0x18 | 0x104 | 16-bit | Debug Stopped | W | Register R4 |
| 0x18 | 0x105 | 16-bit | - | R | Register R5 |
| 0x18 | 0x105 | 16-bit | Debug Stopped | W | Register R5 |
| 0x18 | 0x106 | 16-bit | - | R | Register R6 current mode |
| 0x18 | 0x106 | 16-bit | Debug Stopped | W | Register R6 current mode |

| *Control [5:0]* | *Address* | *Data* | *Required XAP4 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x18 | 0x107 | 16-bit | - | R | Register R7 current mode |
| 0x18 | 0x107 | 16-bit | Debug Stopped | W | Register R7 current mode |
| 0x18 | 0x110 | 16-bit | - | R | Register R0_U |
| 0x18 | 0x110 | 16-bit | Debug Stopped | W | Register R0_U |
| 0x18 | 0x116 | 16-bit | - | R | Register R6_U |
| 0x18 | 0x116 | 16-bit | Debug Stopped | W | Register R6_U |
| 0x18 | 0x117 | 16-bit | - | R | Register R7_U |
| 0x18 | 0x117 | 16-bit | Debug Stopped | W | Register R7_U |
| 0x18 | 0x120 | 16-bit | - | R | Register R0_S |
| 0x18 | 0x120 | 16-bit | Debug Stopped | W | Register R0_S |
| 0x18 | 0x126 | 16-bit | - | R | Register R6_S |
| 0x18 | 0x126 | 16-bit | Debug Stopped | W | Register R6_S |
| 0x18 | 0x127 | 16-bit | - | R | Register R7_S |
| 0x18 | 0x127 | 16-bit | Debug Stopped | W | Register R7_S |
| 0x18 | 0x130 | 16-bit | - | R | Register R0_I |
| 0x18 | 0x130 | 16-bit | Debug Stopped | W | Register R0_I |
| 0x18 | 0x136 | 16-bit | - | R | Register R6_I |
| 0x18 | 0x136 | 16-bit | Debug Stopped | W | Register R6_I |
| 0x18 | 0x137 | 16-bit | - | R | Register R7_I |
| 0x18 | 0x137 | 16-bit | Debug Stopped | W | Register R7_I |
| 0x18 | 0x140 | 16-bit | - | R | Register R0_N |
| 0x18 | 0x140 | 16-bit | Debug Stopped | W | Register R0_N |
| 0x18 | 0x146 | 16-bit | - | R | Register R6_N |
| 0x18 | 0x147 | 16-bit | Debug Stopped | W | Register R6_N |
| 0x18 | 0x200 | 16-bit | - | R | Register BRK0 |

| *Control [5:0]* | *Address* | *Data* | *Required XAP4 SIF Processor Modes* | *R/W* | *Meaning* |
|---|---|---|---|---|---|
| 0x18 | 0x200 | 16-bit | Debug Stopped | W | Register BRK0 |
| 0x18 | 0x300 | 16-bit | - | R | Register FLAGS |
| 0x18 | 0x300 | 16-bit | Debug Stopped | W | Register FLAGS |
| 0x18 | 0x301 | 16-bit | - | R | Register SFLAGS_S |
| 0x18 | 0x301 | 16-bit | Debug Stopped | W | Register SFLAGS_S |
| 0x18 | 0x302 | 16-bit | - | R | Register SFLAGS_I |
| 0x18 | 0x302 | 16-bit | Debug Stopped | W | Register SFLAGS_I |
| 0x18 | 0x303 | 16-bit | - | R | Register SFLAGS_N |
| 0x18 | 0x303 | 16-bit | Debug Stopped | W | Register SFLAGS_N |
| 0x18 | 0x304 | 16-bit | - | R | Register BRKE |
| 0x18 | 0x304 | 16-bit | Debug Stopped | W | Register BRKE |
| 0x18 | 0x400 | 16-bit | - | R | Register PC |
| 0x18 | 0x400 | 16-bit | Debug Stopped | W | Register PC |

### 9.6.3 XAP4 Status Register

One of the SIF Debug Instructions reads the selected XAP4 Status Register. Note that this is not the same as the SIF shift register Status field, STATUS[11:0].

The XAP4 Status Register contains the following bit fields :

| *Bit* | *Name* | *0 Meaning* | *1 Meaning* |
|---|---|---|---|
| 0 | DEBUG | XAP4 is in Debug mode | XAP4 is in Normal mode |
| 1 | WAKE_UP | wake_up input = 0 | wake_up input = 1 |
| 2 | FORCE_STOP | force_stop input = 0 | force_stop input = 1 |
| 3 | FORCE_STOPPED | force_stop input has not been high since last SIF instruction (xSIF or iSIF). | force_stop input has been high since last SIF instruction (xSIF or iSIF). |
| 4 | MODE0 | MODE[1:0] = M[1:0] bits in XAP4 Flags Register. See section 5.7<br>MODE[1:0] indicates the current processor mode of the XAP4 :<br>0 Supervisor mode<br>1 User mode<br>2 Interrupt mode<br>3 NMI mode | |
| 5 | MODE1 | | |
| 6 | BRK_FLAG | = B bit in XAP4 Flags Register. See section 5.7 | |
| 7 | STEP_FLAG | = T bit in XAP4 Flags Register. See section 5.7 | |
| 8 | RUN_STATE0 | RUN_STATE[1:0] indicates the current run state of the XAP4. This is not affected by the force_stop input.<br>0 Run Continuous<br>1 Run to Breakpoint<br>2 Stopped<br>3 Single Step | |
| 9 | RUN_STATE1 | | |

| *Bit* | *Name* | *0 Meaning* | *1 Meaning* |
|---|---|---|---|
| 10 | SLEEPNOP | XAP4 is not executing a sleepnop instruction. | XAP4 is executing a sleepnop instruction. |
| 11 | SLEEPSIF | XAP4 is not executing a sleepsif instruction. | XAP4 is executing a sleepsif instruction. |
| 12 | XSIF_ACTIVITY | An xSIF instruction has not been started since the last iSIF instruction. | An xSIF instruction has been started since the last iSIF instruction. |
| 13 | ISIF_ACTIVITY | An iSIF instruction has not been started since the last xSIF instruction. | An iSIF instruction has been started since the last xSIF instruction. |
| 14 | | | |
| 15 | | | |

## 9.7 SIF Error Processing

### *9.7.1 SIF Parity Bits*

Normally parity bits can only expose single bit errors. However, the XAP4 SIF has a parity-checking configuration which also detects stuck high and stuck low faults on SIF_MOSI and SIF_MISO.

#### 9.7.1.1 PARITY_AC

PARITY_AC (Address + Control field parity) is set by the SIF Master and checked by the SIF Slave. It has odd parity, so there are an odd number of 1s in the 24 bits :

- CONTROL[7:0]
- ADDRESS[15:0]

If SIF_MOSI is stuck at 1 or stuck at 0, there will be a parity error in PARITY_AC. This is because there is an even number of protected bits (including PARITY_AC) and odd parity checking. This means that the bits cannot all be the same value.

If the Slave detects a parity error in PARITY_AC it should :

- Not pass the SIF Instruction on to a processor.
- Set ERROR = 1 in Status field.
- Set ERROR_TYPE = 0 in the Status field.
- Format ERROR_CODE[3:0] in the Status field.
- Complete the SIF Handshake (SIF_LOADB).

#### 9.7.1.2 PARITY_D

PARITY_D (Data field parity) is set by the SIF Slave and checked by the SIF Computer. It has odd parity, so there are an odd number of 1s in the 17 bits :

- PARITY_D
- DATA[15:0]

PARITY_D is based on the values in the :

- Data Capture Register, for valid SIF Reads (i.e when Data Shift Register is updated with a parallel load from the Data Capture Register).
- Data Shift Register for SIF Writes and SIF Errors (i.e when Data Shift Register is not updated with a parallel load from the Data Capture Register).

#### 9.7.1.3 PARITY_S

PARITY_S (Status field parity) is set by the SIF Slave and checked by the SIF Computer. PARITY_S should be calculated after PARITY_D. It has odd parity, so there are an odd number of 1s in the 12 bits :

- STATUS[11:0]

If SIF_MISO is stuck at 1 or stuck at 0, there will be a parity error in PARITY_S. This is because there is an even number of protected bits (including PARITY_S) and odd parity checking. This means that the bits cannot all be the same value.

PARITY_S is based on the values in the Status Capture Register (which will always be parallel loaded into the Status Shift Register when a SIF Instruction is completed).

### *9.7.2 SIF Error Reporting in Status Field*

#### 9.7.2.1 SIF Error Codes

When the SIF detects an error in the SIF Instruction it should set :

- ERROR = 1
- ERROR_TYPE = 0
- ERROR_CODE[3:0] as follows :

| *SIF Error Code* | *Description* |
|---|---|
| 0 | Parity error in PARITY_AC |
| 1 | Unrecognised Direct SIF Instruction |
| 2 | Wrong data-value in Direct SIF Instruction |
| 3 | Unrecognised Processor |
| 4 | iSIF Instruction Cancelled |

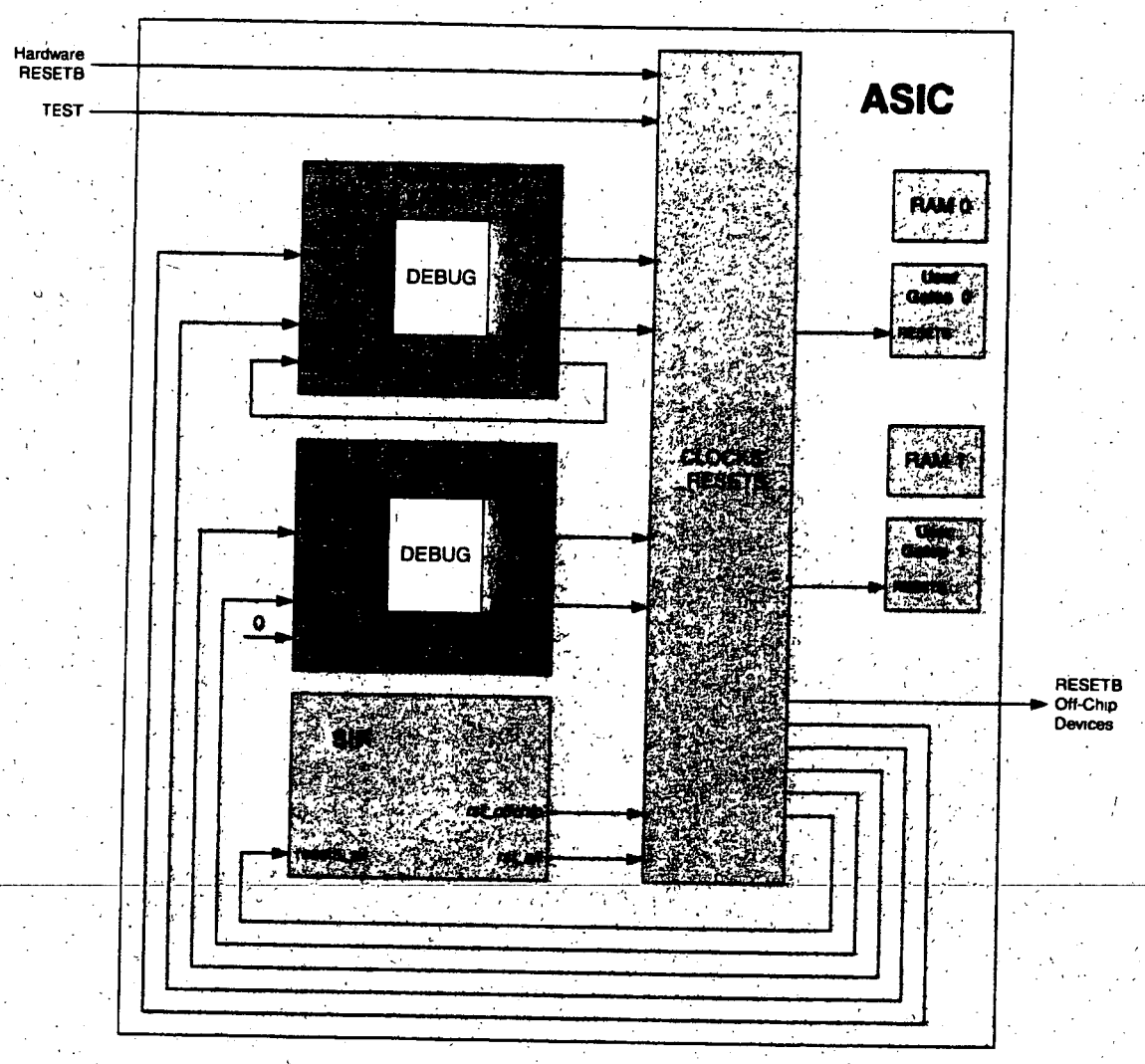
#### 9.7.2.2 Processor Error Codes

When the selected XAP4 detects an error in the SIF Instruction it should set :

- ERROR = 1
- ERROR_TYPE = 1
- ERROR_CODE[3:0] as follows :

| *Processor Error Code* | *Description* |
|---|---|
| 0 | Unrecognised SIF Instruction |
| 1 | Wrong data-value in SIF Instruction |
| 2 | Invalid SIF Write |
| 3 | Invalid SIF Read |
| 4 | Processor not in Debug mode |
| 5 | Processor not Stopped |

## 9.8 SIF Resets




The SIF and XAP4 Verilog modules contain :

- resetb_* - Asynchronous active low reset inputs (all flipflops will be cleared even if clocks are stopped). By default, all flipflops are reset to 0.
- rst_* - Synchronous (clocked by system CLK) active high reset outputs.

The SIF and XAP4 Verilog modules do not contain any test mode circuitry. They do not impose or assume any test strategy for the ASIC. They are scannable designs that the designer can configure for a variety of test strategies.

All conditioning of clock and reset signals as required for Scan and other test modes should be done in the CLOCKS_RESETS module. This module generates the resetb resets that are fed to the various modules in the ASIC. These signals will often need to be controlled by external pins in TEST mode, to make the ASIC fully scannable. The

CLOCKS_RESETS module is application-specific and should be designed by the user to conform to the test strategy being used for the ASIC.

The SIF Instructions include several resets for different parts of the ASIC and Off-Chip devices. SIF resets are done by 2 mechanisms :

- Direct SIF Instructions. Reset 'everything', off-chip devices, whole ASIC, SIF itself.
- Processor SIF Instructions to a selected processor. Reset that processor, associated user gates

To avoid resets occuring by accident :

- Direct SIF Instruction resets require special data values.
- Processor SIF Instructions require the selected processor to be in Debug mode.

The XAP4 has 2 reset inputs :

- resetb_xap. Resets all XAP4 except Debug state. Most flipflops are set to 0.
- resetb_sif. Resets SIF and Debug state of all XAPs. Most flipflops are set to 0.

The flipflops in XAP4 are either reset by resetb_xap or by resetb_sif, but never by both.

resetb_xap is often pulled low by xIDE executing the relevant Debug SIF Instruction. When resetb_xap is pulled low, the XAP will be reset to :

- MODE[1:0] = 00 = Supervisor Mode

resetb_sif is normally only pulled low at power-on-reset. When resetb_sif is pulled low, the XAP will be reset to :

- DEBUG = 0 = Normal Mode
- RUN_STATE[1:0] = 00 = Running

If the designer wants the XAP to be stopped after resetb_sif is pulled low, he should connect the self_force_stop output to the force_stop input. In the diagram on the previous page, Processor 0 is stopped and Processor 1 is running after resetb_sif is pulled low.

-822-

Cambridge Consultants

# XAP4

## Programmer's Manual

**Cambridge Consultants Limited**
Science Park
Milton Road
Cambridge
England
CB4 0DW

+44 (0) 1223 420024
xap@CambridgeConsultants.com
www.CambridgeConsultants.com

**C7432-UM-002 v1.3**
**29 November 2005**

# Revision History

| | | |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

Cambridge Consultants

# Legal Notices

This is the Programmer's Manual for the XAP4 processor. You may use this if you accept the following conditions. If you do not accept these conditions, you must delete or dispose of this copy of the XAP4 Programmer's Manual immediately.

Copying. You are free to copy this manual within your organisation, but you must not copy it in whole or in part to anyone outside your organisation without the prior written agreement of Cambridge Consultants Ltd. This manual can be downloaded from www.CambridgeConsultants.com/ASIC by those who accept these conditions.

Fit for purpose: The XAP4 processor and the xIDE software development environment must not be used for safety critical and/or life support applications without a specific written agreement from Cambridge Consultants Ltd.

Liability Cambridge Consultants Ltd. makes no warranty of any kind with regard to the information contained in this manual, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Cambridge Consultants Ltd. shall not be liable for omissions or errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material. The information contained herein may be updated from time to time.

Ownership and Licence: The XAP4 processor (including associated technologies including SIF), XAP4 documentation and the xIDE software development environment and all intellectual property rights associated with them are owned by Cambridge Consultants Ltd and are protected by copyright law and international copyright treaties, patent applications, letters patent and other intellectual property laws and treaties. You must not attempt to use the XAP4 processor or the xIDE developer's environment unless you have a valid licence agreement with Cambridge Consultants Ltd. You must not use the information within this manual or other items supplied to you for the purposes of designing, developing or testing a device or simulator which is, or is intended to be, wholly or partly instruction set compatible with the XAP4 processor. You are authorised to open, view and print any electronic copy we send you of this manual within your organisation. Printouts of this manual must be kept within your organisation.

Copyright and trademarks: This manual is © copyright Cambridge Consultants 2005. Cambridge Consultants Ltd., the Cambridge Consultants logo, XAP, XAP1, XAP2, XAP3, XAP4, XAP5 and xIDE are trademarks of Cambridge Consultants Ltd. All other trademarks mentioned herein are the property of their respective owners.

Cambridge Consultants Ltd
Science Park
Milton Rd
Cambridge CB4 0DW
England

www.CambridgeConsultants.com/ASIC

© Cambridge Consultants Ltd 2005

# Project Team

Cambridge Consultants

-825-

# Table of Contents


1 INTRODUCTION ..... 6

1.1 THIS DOCUMENT ..... 6
1.2 ASSUMPTIONS ..... 6
1.3 ABOUT XAP4 ..... 6
1.4 THE XAP FAMILY ..... 8
1.5 XAP4 SYSTEMS ..... 9
1.6 OTHER DOCUMENTATION ..... 10
1.7 FOR FURTHER HELP ..... 11
1.8 GLOSSARY AND ACRONYMS ..... 11

2 PROGRAMMER'S MODEL ..... 13

2.1 PROCESSOR STATES ..... 13
2.2 PROCESSOR MODES ..... 14
*2.2.1 Capabilities of the processor modes* ..... 15
*2.2.2 Typical usage of processor modes* ..... 16
2.3 REGISTERS ..... 16
*2.3.1 Normal registers* ..... 16
*2.3.2 Program Counter* ..... 18
*2.3.3 Special registers* ..... 18
*2.3.4 Breakpoint register* ..... 20
2.4 PIPELINE ..... 21
2.5 MEMORY ARCHITECTURE ..... 21
2.6 STACK OPERATION ..... 21
2.7 RESET ..... 22
2.8 INTERRUPTS AND EXCEPTIONS ..... 22
*2.8.1 Exception processing in User Mode* ..... 23
*2.8.2 Exception processing in the privileged modes* ..... 23
*2.8.3 Interrupt processing* ..... 24
*2.8.4 Returning from interrupts and exceptions* ..... 25
*2.8.5 Nested interrupts* ..... 25
*2.8.6 Interrupt vector table* ..... 26
*2.8.7 Exceptions* ..... 27
2.9 DEBUGGING ..... 31

3 C LANGUAGE INTERFACE ..... 34

3.1 DATA TYPES ..... 34
3.2 DATA ALIGNMENT ..... 34
*3.2.1 Aligned data* ..... 35
*3.2.2 Unaligned data* ..... 35
*3.2.3 Methods of accessing unaligned data* ..... 35
3.3 CALLING CONVENTION ..... 36
3.4 APPLICATION MODELS ..... 37
*3.4.1 Static linkage* ..... 37
*3.4.2 Dynamic linkage* ..... 37

4 INSTRUCTION SET OVERVIEW ..... 39
4.1 SUMMARY OF ASSEMBLER SYNTAX ..... 39
4.1.1 Instruction mnemonics ..... 39
4.1.2 Operands ..... 39
4.1.3 Registers ..... 40
4.1.4 Register Lists ..... 40
4.1.5 Comments ..... 40
4.1.6 Number formats ..... 40
4.1.7 Labels ..... 41
4.1.8 Directives ..... 42
4.2 INSTRUCTION ENCODING ..... 42
4.3 ADDRESSING MODES ..... 44
5 INSTRUCTION GROUPS ..... 44
5.1.1 Branches ..... 44
5.1.2 Load and Store ..... 45
5.1.3 Push and Pop ..... 48
5.1.4 Move ..... 49
5.1.5 ALU operations ..... 50
5.1.6 Compare operations ..... 50
5.1.7 Shift and rotate ..... 50
5.1.8 Block operations ..... 52
5.1.9 Miscellaneous instructions ..... 53
5.2 COMMON CODE SEQUENCES ..... 53
5.2.1 Function prologue and epilogue ..... 54
5.2.2 Nested interrupt prologue and epilogue ..... 54
5.2.3 Semaphore ..... 55
5.2.4 Initialisation of User Mode registers ..... 56
5.3 INSTRUCTIONS GROUPED BY FUNCTION ..... 56
5.3.1 Branches ..... 56
5.3.2 Load and Store ..... 57
5.3.3 Push and Pop ..... 57
5.3.4 Move ..... 58
5.3.5 ALU operations ..... 58
5.3.6 Compare operations ..... 59
5.3.7 Shift and rotate ..... 59
5.3.8 Block copy and store ..... 60
5.3.9 Miscellaneous instructions ..... 60
5.4 ALPHABETICAL LIST OF ALL INSTRUCTIONS ..... 61
5.5 PRIVILEGED INSTRUCTIONS ..... 66
5.6 ALIASED INSTRUCTIONS ..... 67
5.7 REGISTER NAMING IN INSTRUCTIONS ..... 67
5.8 REGISTER SPECIFICATION FIELDS ..... 67
5.9 IMMEDIATES ..... 69
5.10 REGISTER LIST FIELDS ..... 70
6 INSTRUCTION SET REFERENCE ..... 72

# 1 Introduction

## 1.1 This Document

This document is the primary reference for software engineers developing programs to run on XAP4 systems. It contains all the information needed to understand the XAP4 architecture from a software perspective.

Further documents describe other aspects of XAP4 systems. Refer to section 1.6, "Other documentation" for details.

## 1.2 Assumptions

The reader is assumed to be familiar with microprocessor architectures in general, and to have some understanding of high level languages such as C. A detailed understanding of digital electronics is not required.

## 1.3 About XAP4

The XAP4 is a powerful 16-bit processor optimised for low gate count and low power. It has a von Neumann architecture and can address 64KByte of linear byte-addressable memory.

### *High Code Density*

The XAP4 architecture, instruction set, compiler toolchain and application binary interface are all designed for efficient execution of programs written in C.

### Complex Systems

The XAP4 implements the features necessary to support complex software systems where high reliability or high integrity is a requirement. The XAP4 provides hardware support for common real time operating system primitives, including a clear definition of user and supervisor modes, atomic instructions for synchronisation constructs, and position independent code to allow dynamic linkage.

The XAP4 MMU implements memory protection systems for instruction and data, throwing exceptions if processes access code or data memory illegally.

### Interrupts and Exceptions

The XAP4 supports 16 interrupts and 16 exceptions. Each has its own handler defined in a vector table.

### Rich RISC-like Instruction Set

A rich set of instructions is provided, including a complete set of branches, arithmetic operations on 8, 16 and 32 bit data. Several instructions map closely onto C language constructs thereby providing very high code density and fast execution.

### 16-bit and 32-bit Instructions

The XAP4 instruction set contains 145 unique instructions. Most common instructions are 16 bits long with less common instructions using 32 bits. Some instructions have 16-bit and 32-bit forms, with the toolchain using the 16-bit form whenever possible.

This approach gives high code density. Programs can mix 16-bit and 32-bit instructions at will. No switching between modes is necessary and the processor executes all instructions at full speed. The 16-bit instructions can access all of the processor resources available to the 32-bit instructions.

### Hardware Acceleration

The processor includes a fast multiplier, divider and barrel shifter. Instruction execution is pipelined, resulting in operating speeds of up to 100MHz on ASIC processes between 0.7µm and 0.09µm.

### Targeted for Speed or Size

XAP4a is targeted for low cost applications and requires approximately 9000 gates. Other implementations of the XAP4 architecture will be available in the future which target speed over size.

### Language Support

The XAP4 GNU Compiler Collection (gcc) provides an industry standard ANSI C compiler.

### Development Tools

The xIDE integrated development environment provides a cross-platform development and debugging tool for XAP4 developers, using an industry-standard look and feel.

## 1.4 The XAP Family

The XAP4 is the fourth member of the Cambridge Consultants XAP family of embedded processors for ASICs and FPGAs. All four processors use the Cambridge Consultants SIF interface for debugging and the xIDE development environment for software development.

### Legacy Processors

The XAP1 is targeted at extremely small or extremely low power applications. It is a 16-bit Harvard architecture processor and fits in approximately 3000 gates. The XAP1 has been used in various ASIC projects since 1994.

The XAP2 is an evolutionary development from XAP1. It is again a 16-bit Harvard architecture processor but has a larger address space and better optimisation for C language applications. It requires approximately 12,000 gates. It was developed in 1999 and is used in many high-volume products.

### Current Processors

The XAP3 is a completely new development which maintains the strategy of low power, low cost and low risk, whilst providing the size and sophistication to run complex applications. It is a 32-bit von Neumann processor and fits in approximately 30,000 gates. Variants of the XAP3 architecture are available which specifically target speed or size. The XAP3 supports C and C++.

The XAP4 is a successor to the XAP3 and shares many of its architectural features to maintain the strategy of low power, low cost and low risk.

XAP5, like XAP4, is an advanced 16-bit processor but can address 16MByte of memory and requires approximately 12,000 gates.

-830-

## 1.5 XAP4 Systems

The XAP4 exists as a soft IP core written entirely in Verilog. This can be targeted at either system-on-chip (ASIC) or FPGA implementations.

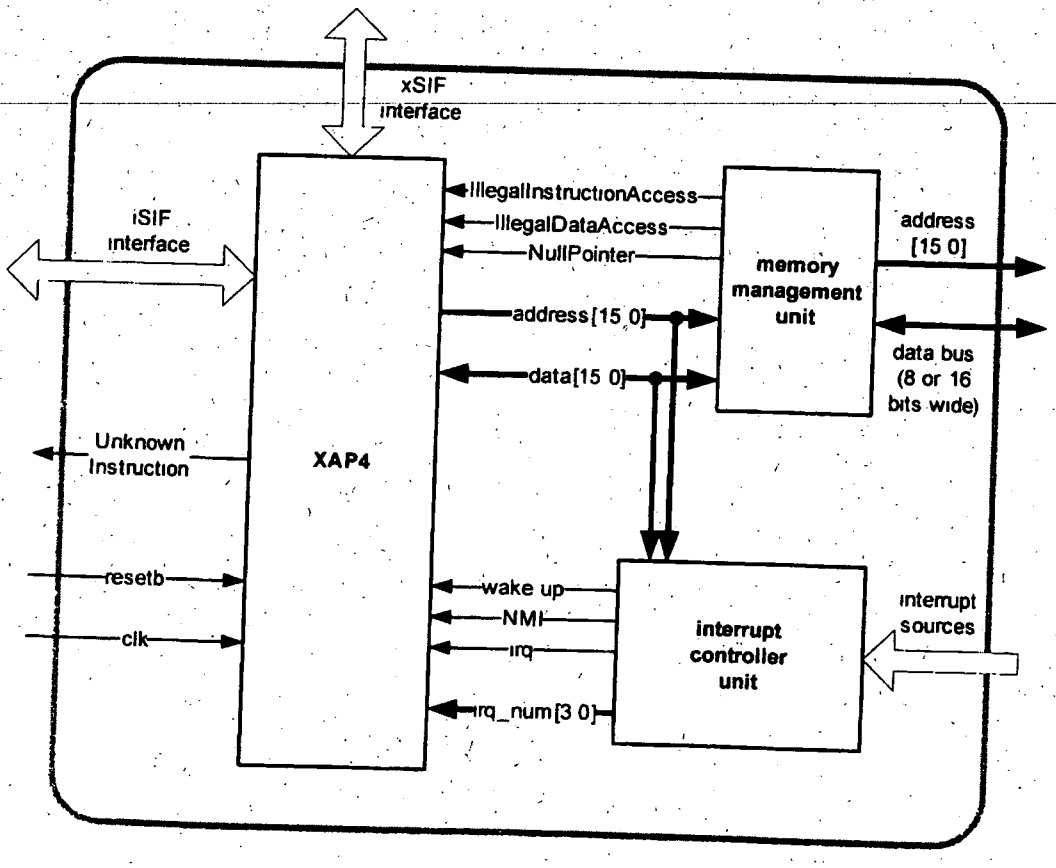
The xIDE toolset includes an instruction set simulator for XAP4. This can be extended to include models of other parts of the system, including memories and interrupts.

The XAP4 normally exists with other standard components in a system. These are:

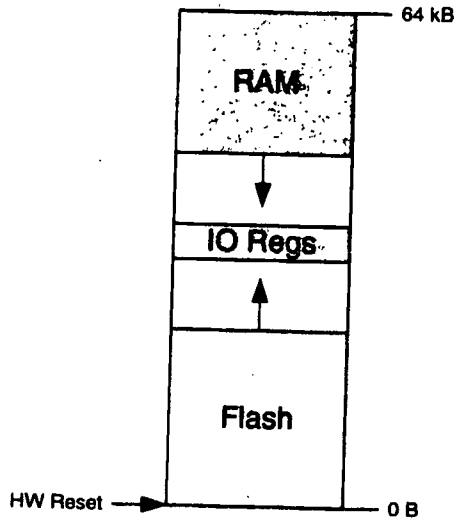
- A Memory Management Unit (MMU) to interface with memories, typically flash and RAM.
- An Interrupt Controller Unit (ICU) to prioritise interrupt sources and provide an interrupt number to the XAP4.

Application-specific circuitry is needed for:

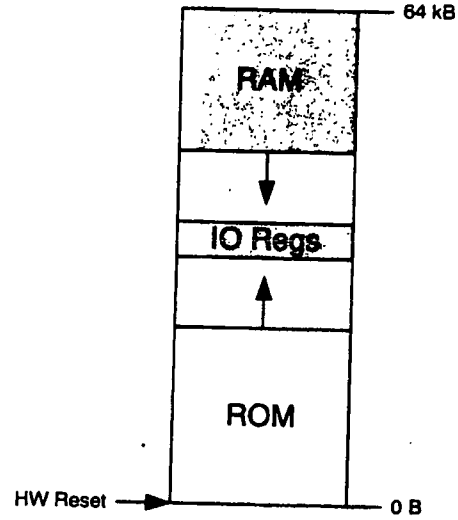
- System control - clock and reset generation, watchdog functions.
- Application-specific memories, peripherals and interrupt sources.



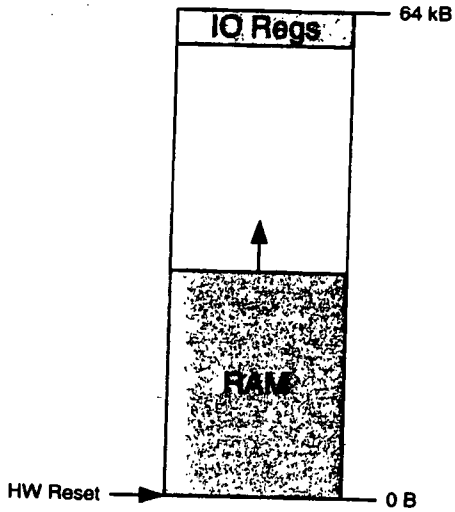
The figures below show the recommended memory maps for the four most common configurations.



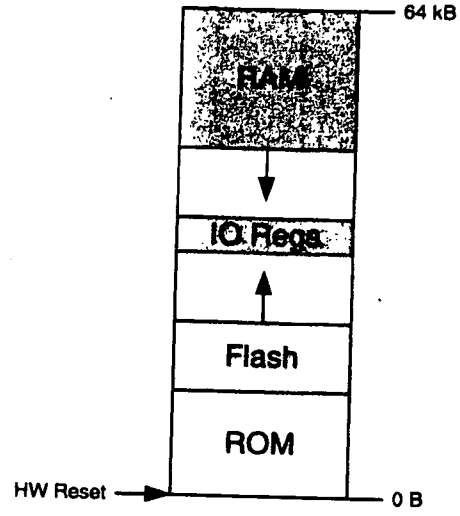
RAM + Flash Configuration



RAM + ROM Configuration



RAM Only Configuration



RAM + ROM + Flash Configuration

## 1.6 Other Documentation

The following documents describe the software tools and xIDE integrated development environment:

| Document | Contains |
|---|---|
| xIDE User Manual C7066-TM-001 | Describes the general features of xIDE. |
| xIDE Python Object Model C7066-TM-003 | Describes the python object model used within xIDE. This is essential for developers wanting to use the scripting features of xIDE. This allows developers to perform automated testing or to |

| | |
|---|---|
| | model other parts of a system during simulation. |
| xIDE for XAP4<br>C7066-TM-017 | Describes specific features of the XAP4 plugin for xIDE. |
| XAP4 GNU Binutils Manual<br>C7432-UM-003 | Describes the XAP4 Assembler, Linker and other binary utilities. |
| XAP4 GCC Compiler Manual<br>C7432-UM-004 | Describes the XAP4 ANSI C compiler |
| XAP4a Instruction Set Quick Reference Card<br>C7432-UM-005 | A short guide to the XAP4 instruction set. |

For details of the XAP4 hardware, refer to these documents:

| Document | Contains |
|---|---|
| XAP4a Hardware Reference Manual<br>C7432-UM-011 | Information needed by digital designers using the XAP4a in an FPGA or ASIC |
| XAP4a Hardware Quickstart Guide<br>C7245-UM-001 | A short guide on how to get started in synthesis and simulation of XAP4a. |
| XAP4 Datasheet ASICs-SB-011 | A short overview of the XAP4 architecture. |

## 1.7 For Further Help

The XAP4 documentation set provides answers to most questions.

If a problem remains unsolved, contact Cambridge Consultants technical support at:

xap@CambridgeConsultants.com

## 1.8 Glossary and Acronyms

| Term | Meaning |
|---|---|
| Double-Word | A 4-byte (or 32-bit) quantity. |
| Word | A 2-byte (or 16-bit) quantity. The XAP4 is a 16-bit processor. |
| Byte | An 8-bit quantity. |
| Little Endian | A system in which the least significant byte of a word is stored at the lowest memory address. The XAP4 is a little endian processor. |

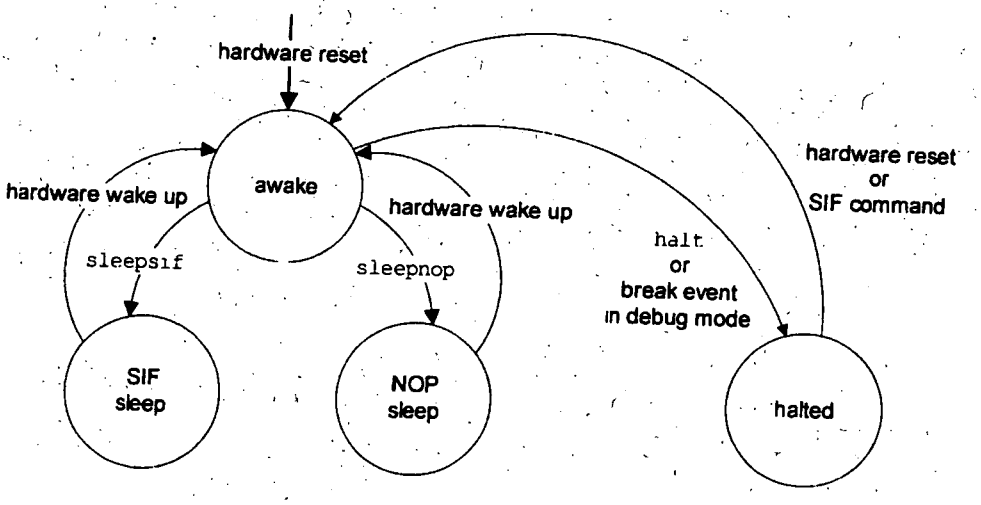
| | |
|---|---|
| Big Endian | A system in which the most significant byte of a word is stored at the lowest memory address. |
| ALU | Arithmetic and Logic Unit. |
| ICU | Interrupt Controller Unit. |
| MMU | Memory Management Unit. |
| Set | In connection with the flags, set indicates a logic 1 in the flag |
| Clear | In connection with the flags, clear indicates a logic 0 in the flag |

# 2 Programmer's Model

## 2.1 Processor States

The XAP4 has four processor states:

| Processor State | Description |
|---|---|
| Awake | The XAP4 is able to execute instructions. This is the XAP4 processor state following a reset or after a wake up from one of the sleep states. |
| SIF Sleep | The XAP4 is asleep but SIF cycles can still occur. This state is entered with the `sleepsif` instruction. |
| NOP Sleep | The XAP4 is asleep and SIF cycles are not permitted. This state is entered with the `sleepnop` instruction. |
| Halted | The XAP4 is stopped. This state is entered with the `halt` instruction or after a `Break` event when in debug mode. Refer to section 2.8.7, "Break" for details of break events. |



Returning to the awake state from any other state requires hardware activity. Power consumption is significantly reduced in the two sleep states.

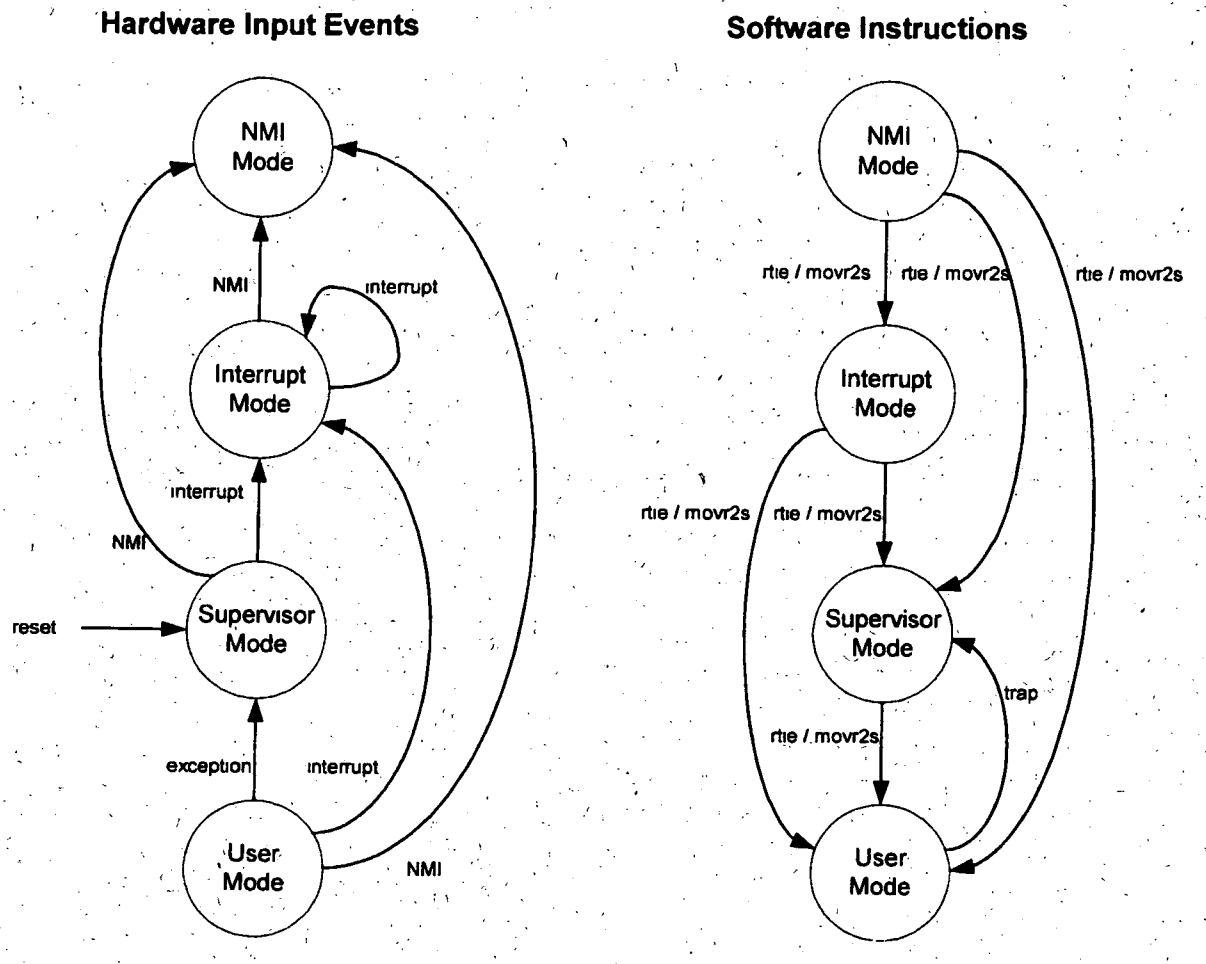
## 2.2 Processor Modes

The XAP4 has four processor modes:

| Processor Mode | | Description |
|---|---|---|
| User Mode | | Code running in User Mode cannot affect the operation of code running in the other modes. Some instructions are not permitted in User Mode. Conversely, all code in User Mode can be controlled from Supervisor Mode.<br>User Mode is suitable for untrusted application code. |
| Privileged Modes | Supervisor Mode | Code running in Supervisor Mode can execute all the XAP4 instructions and can therefore control the activity of code running in User Mode.<br>The XAP4 enters Supervisor Mode after a reset or on execution of a trap instruction.<br>Supervisor Mode is suitable for operating systems. |
| | Interrupt Mode | Interrupt Mode is entered on a hardware interrupt. |
| | NMI Mode | NMI Mode is entered on a non-maskable interrupt. |

The Supervisor, Interrupt and NMI Modes are the privileged modes.

The diagram below illustrates how mode changes take place:




It is possible to change from NMI, Interrupt and Supervisor Mode to any of the four modes by using the `movr2s` instruction. The diagram shows only the mode changes expected in normal interrupt and exception handlers.

### 2.2.1 Capabilities of the processor modes

Code running in User Mode can only access the User Mode registers and cannot write to the FLAGS register. Code running in any of the privileged modes can access its own registers and the User Mode registers.

Code running in Supervisor Mode cannot access Interrupt or NMI Mode registers, and vice versa. In order to initialise Interrupt and NMI Mode registers, it is necessary to disable interrupts and perform mode changes by writing to the FLAGS register with the `movr2s` instruction.

| | | current processor mode | | | |
|---|---|---|---|---|---|
| | | **User** | **Supervisor** | **Interrupt** | **NMI** |
| **can access** | **User** | yes | yes | yes | yes |
| | **Supervisor** | no | yes | no | no |

| | | | | | |
|---|---|---|---|---|---|
| these registers | Interrupt | no | no | yes | yes (Interrupt Mode R7_I only) |
| | NMI | no | no | no | yes |

### 2.2.2 Typical usage of processor modes

#### *Simple applications*

Simple applications may execute in Supervisor Mode and make no use of User Mode. Interrupt Mode and NMI Mode are used for interrupt and NMI operations.

#### *Complex applications*

For complex applications it is likely that most of the processing will occur in User Mode. Services provided by an operating system will execute in Supervisor Mode. User Mode code can call Supervisor Mode code using the `trap` instructions.

## 2.3 Registers

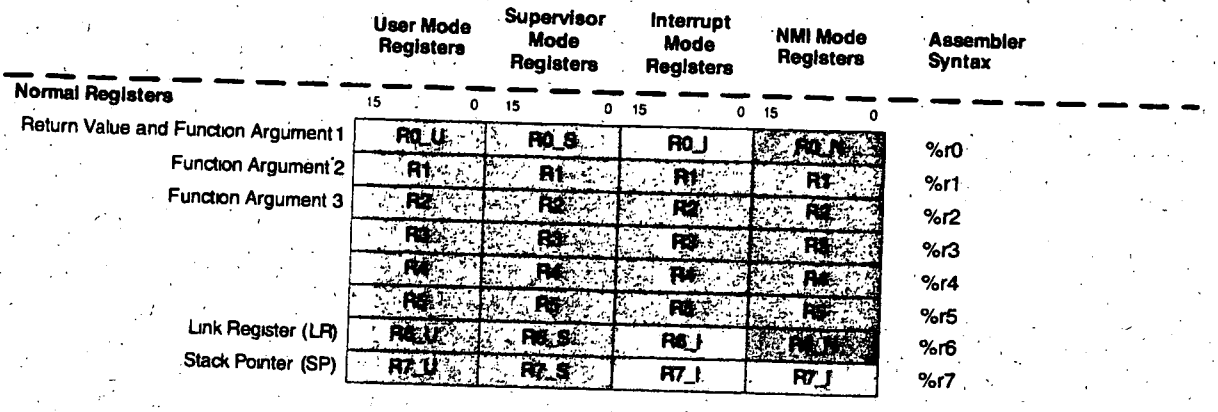
The XAP4 register set includes:

- 8 Normal registers called R0 to R7. Three of the normal registers are shadowed in hardware to assist with processor mode switches.
- The Program Counter, called PC.
- 5 Special registers. These are FLAGS, SFLAGS_S, SFLAGS_I, SFLAGS_N and BRKE.
- 1 Breakpoint register, called BRK0.

In the descriptions of the registers which follow, hardware registers are in upper case (for example, R1) whilst the assembler name for a register is in lower case (for example, `%r1`).

### 2.3.1 Normal registers

The XAP4 has eight normal registers, called R0 to R7. Registers R0, R6 and R7 are shadowed in hardware such that the hardware register used depends on the current processor mode.




The diagram shows which registers are used in each of the modes. The privileged modes are able to access R0_U, R6_U and R7_U by using the `movr2s` and `movs2r` instructions.

Some of the normal registers are allocated dedicated functions in the XAP4 architecture:

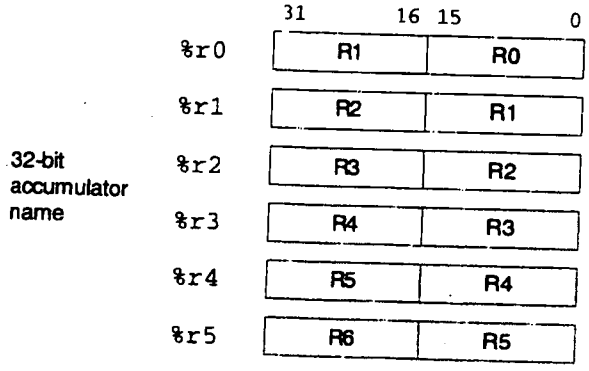
| Assembler syntax | Hardware register name | Notes |
|---|---|---|
| `%r0` | R0_U or R0_S or R0_I or R0_N | The hardware register used as `%r0` depends on the processor mode.<br>To refer specifically to R0_U (in the `movs2r` and `movr2s` instructions), use `%r0_u`. |
| `%r1` | R1 | |
| `%r2` | R2 | |
| `%r3` | R3 | |
| `%r4` | R4 | |
| `%r5` | R5 | |
| `%r6` | R6_U or R6_S or R6_I or R6_N | The hardware register used as `%r6` depends on the processor mode.<br>R6 is used as a Link Register during subroutine calls.<br>To refer specifically to R6_U (in the `movs2r` and `movr2s` instructions), use `%r6_u`. |
| `%r7` | R7_U or R7_S or R7_I | The hardware register used as `%r7` depends on the processor mode. R7_I is used for Interrupt and NMI Modes.<br>R7 is used as a Stack Pointer. |

| | | |
|---|---|---|
| | | To refer specifically to R7_U (in the `movs2r` and `movr2s` instructions), use `%r7_u`. |

### *32-bit accumulators*

For some instructions, the XAP4 uses adjacent pairs of 16-bit registers to create a 32-bit accumulator. The 32-bit accumulator is referenced by the name of the lower register.

The accumulators available are:



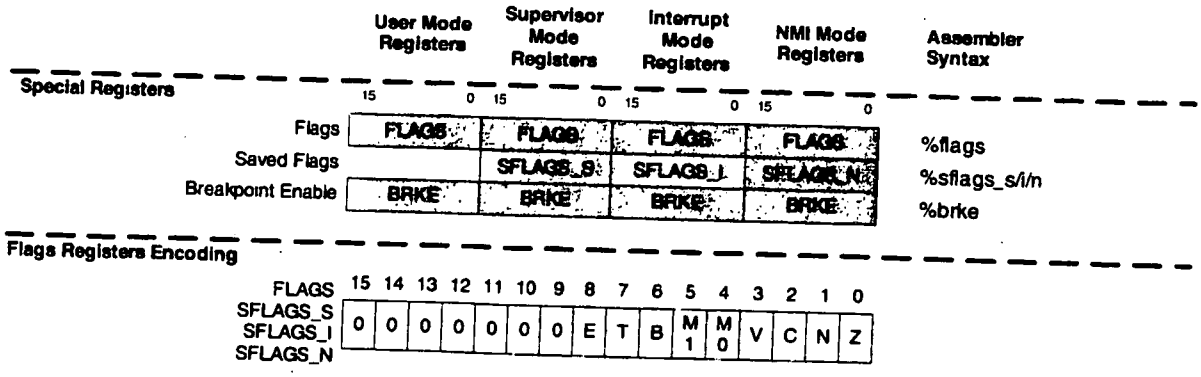

The 32-bit accumulators are used in the mov.32.r, `div.32`, `divrem.32`, `mult.32`, `rem.32`, `rotatel.32`, `shiftl.32`, and `shiftr.32` instructions.

## 2.3.2 Program Counter

The program counter is a 16-bit register that contains the address of the next instruction to be executed. The program counter cannot be accessed directly as a register but is implictly used by all the .p instructions and the `rtie` instruction.

The XAP4 forces the least-significant bit of the program counter to be zero. Attempts to set the least significant bit throw an `IllegalCodeAddress` exception.

## 2.3.3 Special registers



The `movr2s` and `movs2r` instructions allow the privileged modes access to all the special registers. The special registers are described in the following table.

| Assembler syntax | Hardware register Name | Notes |
|---|---|---|
| `%flags` | FLAGS | The flags register. This is used in all processor modes. |
| `%sflags_s` | SFLAGS_S | Saved flags. The processor copies FLAGS into SFLAGS_S when an exception occurs, and restores FLAGS from SFLAGS_S when an `rtie` instruction is executed in Supervisor Mode. |
| `%sflags_i` | SFLAGS_I | Saved flags. The processor copies FLAGS into SFLAGS_I when an interrupt occurs, and restores FLAGS from SFLAGS_I when an `rtie` instruction is executed in Interrupt Mode. |
| `%sflags_n` | SFLAGS_N | Saved flags. The processor copies FLAGS into SFLAGS_N when an NMI occurs, and restores FLAGS from SFLAGS_N when an `rtie` instruction is executed in NMI Mode. |
| `%brke` | BRKE | The breakpoint enable register. |

Each register is described in the following sections.

### *Flags register*

The table below describes the format of the FLAGS, SFLAGS_S, SFLAGS_I and SFLAGS_N registers:

| Bit | Flag | Name | Description |
|---|---|---|---|
| 0 | Z | Zero | Updated by most instructions according to the result of their operation. Refer to the individual instruction descriptions in section 6, "Instruction Set Reference" for details of the flag behaviour. |
| 1 | N | Negative | |
| 2 | C | Carry | |
| 3 | V | Overflow | |
| 5:4 | M[1:0] | Mode | Indicates the current processor operating mode:<br>00 Supervisor Mode<br>01 User Mode<br>10 Interrupt Mode<br>11 NMI Mode |
| 6 | B | Break | Controls the processor's response to a `brk` instruction or a hardware breakpoint event. Refer to section 2.8.7, "Break" for further details. |

Cambridge Consultants

| | | | |
|---|---|---|---|
| 7 | T | Single Step | If set, each User Mode instruction that is executed throws the `SingleStep` exception. |
| 8 | E | Interrupt Enable | If set, interrupts are enabled. Non-maskable interrupts (NMI) cannot be disabled.<br>The E bit is cleared when an interrupt or exception occurs. |
| 15:9 | unused | | Read as zero. |

The following events cause the flags register to be updated:

- A reset.
- An instruction which updates the flags.
- Code running in a privileged mode executes a `movr2s` instruction.
- Code running in a privileged mode executes an `irqd` or `irqe` instruction.
- An exception.
- An interrupt or NMI.

### *Saved Flags registers*

The Saved Flags registers (SFLAGS_S, SFLAGS_I, SFLAGS_N) are used to save the contents of the FLAGS register when the processor changes mode. An `rtie` instruction copies one of the saved flags registers into FLAGS to restore the processor states.

To support nested interrupts, the SFLAGS_I register should be saved on the stack.

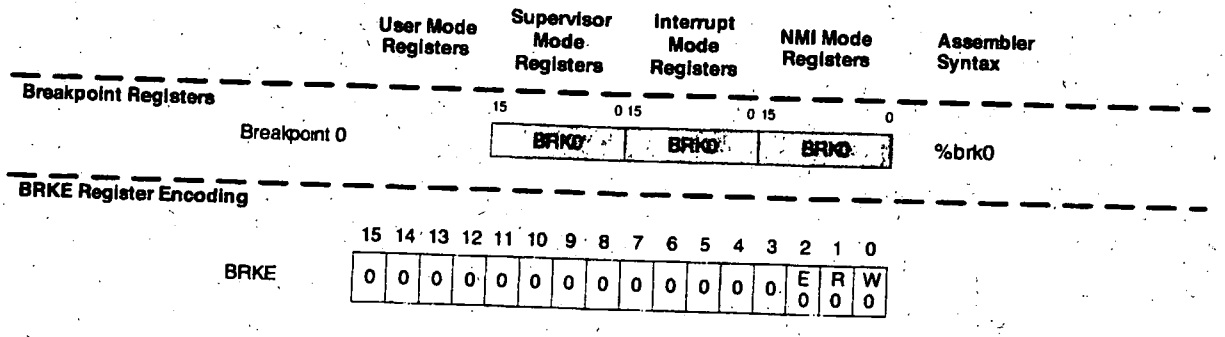
### *Breakpoint Enable register*

The BRKE register is used in conjunction with the breakpoint register. Details are available in the XAP4 Breakpoint and Debug Reference Manual.

## 2.3.4 Breakpoint register

The XAP4 has one breakpoint register and one breakpoint enable register. Details of these are available in the XAP4 Breakpoint and Debug Reference Manual.

The breakpoint register is accessed with the `movr2b` and `movb2r` instructions and is accessible from the privileged modes only.




## 2.4 Pipeline

The XAP4 is a pipelined processor. XAP4a has a two-stage pipeline. For further details, including details of instruction cycle counts, refer to the appropriate Hardware Reference Manual.

## 2.5 Memory Architecture

The XAP4 can address 64KBytes ($2^{16}$ bytes) of memory. The memory space is linear and byte addressable and contains both code and data.

32-bit and 16-bit instructions are aligned to a word (2 bytes) boundary.

For optimum speed and power consumption, the compiler aligns data objects to byte or word boundaries, depending on their type. For more details see the XAP4 GNU C Compiler Manual.

Data objects allocated in assembly code files can be allocated in a packed manner. The XAP4 accesses such unaligned data objects natively and with no programmer effort.

The stacks are aligned to a word boundary (2 bytes).

## 2.6 Stack Operation

The XAP4 stacks start at high addresses and grow downwards in memory. The stacks are 16 bits wide. Push operations decrease the stack pointer by a multiple of two bytes. Pop operations increase the stack pointer by a multiple of two bytes.

Register R7 is used as the stack pointer and points to the last used location on the stack. Separate stacks exist for User Mode, Supervisor Mode and Interrupt Mode.

The compilers operate a fully covered stack. When accessing stack data (local automatic variables or parameters), stack offsets are always positive or zero.

## 2.7 Reset

Following a reset, the XAP4 state is:

- Processor awake.
- Processor Mode is Supervisor.
- All registers are zero.
- All special registers are zero.
- The breakpoint registers are zero.
- The program counter is zero.

The XAP4 fetches the first instruction from address 0, the start of the Interrupt Vector Table.

With the flags register being zero, interrupts are disabled, single stepping is disabled and the `Break` exception is disabled.

## 2.8 Interrupts and Exceptions

Exceptions are generated by code running in User Mode and are generally the result of a `trap` instruction or a problem with the code. In response to the exception, the XAP4 moves to Supervisor Mode and disables interrupts. Once in Supervisor Mode, no further exceptions occur.

Interrupts can occur in any mode whilst interrupts are enabled in the FLAGS register. The XAP4 moves to Interrupt Mode and disables interrupts by clearing the E flaf in the FLAGS register. The XAP4 receives an interrupt number (typically from an external interrupt controller) and branches to an appropriate interrupt handler. At the end of the interrupt processing, the XAP4 returns to the interrupted code with the `rtie` instruction. This restores the interrupt status prior to the interrupt.

Non-maskable interrupts cannot be disabled and can occur in any mode. The XAP4 moves to NMI Mode. At the end of the processing for the non-maskable interrupt, the XAP4 returns to the interrupted code with the `rtie` instruction

Interrupts which occur whilst the XAP4 is in NMI Mode are remembered and are actioned after the exit from NMI Mode with the `rtie` instruction.

### 2.8.1 Exception processing in User Mode

When an exception occurs, the XAP4 sets up the Supervisor Mode registers, disables interrupts, and then branches to the appropriate exception handler. The information in the Supervisor Mode registers on entry to the handler allows the handler to deal with the exception condition in the User Mode code. The Supervisor code may choose to allow the User Mode code to continue, or it may take action to restart the User Mode code.

In detail, the XAP4 exception processing is:

- Store the current flags in SFLAGS_S.
- Store the current PC in R6_S (the Supervisor Mode Link Register). This is the address of the instruction following the instruction which threw the exception.
- Set up R0_S. If the exception was caused by a trap instruction, R0_S is loaded with the argument from the trap instruction. Otherwise, R0_S is loaded with the address of the instruction which caused the exception.
- The XAP4 is put into Supervisor Mode.
- Interrupts are disabled.
- The XAP4 branches to an address in the Interrupt Vector Table. The Interrupt Vector Table would normally contain a branch to an exception handler.

Returning from an exception is done with the `rtie` instruction. If R6_S is not changed by the exception handler, executing an `rtie` causes the User Mode code to resume at the instruction following instruction which threw the exception.

Interrupts are disabled on entry to Supervisor Mode. The exception handler can re-enable them if required with the `irqe` instruction.

### 2.8.2 Exception processing in the privileged modes

Exceptions do not occur in the privileged modes. Exception-generating conditions are treated as follows:

| Exception Condition | Behaviour in the privileged modes |
|---|---|
| `Trap` | A `trap` instruction is treated as a `nop`. |
| `SingleStep` | When single stepping in a privileged mode, this exception does not occur. |

| Exception Condition | Behaviour in the privileged modes |
|---|---|
| `Break` | A break condition (either a `brk` instruction or hardware breakpoint) is treated as a halt or a nop, depending on whether debug mode is enabled or not. Refer to section 2.8.7, "Break" for more details. |
| `UnknownInstruction` | The XAP4 sets an UnknownInstruction output. The system designer can route this signal into the system's interrupt, NMI or reset circuitry. |
| `DivideByZero` | The destination register is not updated.<br>The Z and N flags are updated according to the contents of the destination register.<br>The V flag is set if the operation was signed. The C flag is set if the operation was unsigned. |
| `NullPointer` | Null pointer dereferences are allowed in the privileged modes. |
| `IllegalInstruction` | All instructions are legal in the privileged modes, so this exception cannot happen. |
| `IllegalInstructionAccess` and `IllegalDataAccess` | These conditions are ignored. This allows an operating system to access memory belonging to all tasks. |
| `IllegalCodeAddress` | The least significant bit of the offending address is ignored and the instruction is executed as if the least significant bit was zero. |
| `IllegalStackAddress` | The least significant bit of the stack pointer is ignored and the stack operation is executed as if the least significant bit was zero. |

### 2.8.3 Interrupt processing

The XAP4 has a single interrupt input (IRQ) and a single NMI input. The XAP4 samples the IRQ and NMI inputs at the end of every instruction and during the execution of the block instructions.

If interrupts are enabled, activating the IRQ input causes the XAP4 to switch to Interrupt Mode. The XAP4 then receives an interrupt number which is used to calculate an index into the Interrupt Vector Table. Entries in the Interrupt Vector Table are typically branches to interrupt handlers.

The Non-Maskable Interrupt cannot be disabled by XAP4. Activating the NMI input causes the XAP4 to switch to NMI Mode when the current instruction finishes. The additional shadow registers in NMI Mode ensure that an NMI handler can restore the processor to its state immediately prior to the NMI event.

In detail, the XAP4 interrupt processing is:

- Store the current flags in SFLAGS_I (for an IRQ) or SFLAGS_N (for an NMI).
- Store the current PC in R6_I (for an IRQ) or R6_N (for an NMI). This is the address of the first instruction to be executed on return from the interrupt handler.
- The XAP4 is put into Interrupt or NMI Mode.
- For an IRQ, further IRQs are disabled. For an NMI, further NMIs are not disabled. An IRQ will not take place in NMI Mode because IRQs are not allowed in NMI Mode.
- The XAP4 branches to an address in the Interrupt Vector Table. The Interrupt Vector Table would normally contain a branch to an interrupt handler.

#### *NMI Notes*

The NMI interrupt cannot be disabled. Multiple NMIs may cause the stack to grow if an NMI handler is itself interrupted by another NMI. As NMI Mode and Interrupt Mode share the same stack (R7_I), the Interrupt Mode stack should have sufficient space to allow for an NMI and its stack frame. Care is necessary when using this register in an NMI exception handler.

### 2.8.4 Returning from interrupts and exceptions

The `rtie` instruction is used to return from interrupts and exceptions and is implemented as:

```
if (FLAGS.Mode == NMIMode) then
        FLAGS = SFLAGS_N
        PC    = R6_N
elseif (FLAGS.Mode == InterruptMode) then
        FLAGS = SFLAGS_I
        PC    = R6_I
elseif (FLAGS.Mode == SupervisorMode) then
        FLAGS = SFLAGS_S
        PC    = R6_S
endif
```

Thus the null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code. Restoring the flags causes the E bit (interrupt enable) to be restored to its value before the interrupt or exception occurred.

### 2.8.5 Nested interrupts

Interrupts are disabled on entry to Interrupt Mode. To support nestable interrupts some additional work is required of the interrupt handler to save the processor state prior to re-enabling interrupts with the `irqe` instruction.

The interrupt handler must similarly restore the processor state prior to returning.

Example code for nested interrupts is given in section "Nested interrupt prologue and epilogue".

### 2.8.6 Interrupt vector table

The Interrupt Vector Table (IVT) sits in memory at address 0x0000. Each entry in the IVT is one 32-bit instruction. These are often either:

- `bra.p` to the start of the relevant handler, or
- the null handler (the `rtie` instruction).

The IVT manages both exceptions and interrupts.

The layout of the IVT is shown below. The reserved exceptions are for future expansion.

| Offset into IVT | Exception or Interrupt | Resulting XAP4 mode |
|---|---|---|
| 0x0000 | Reset | Supervisor |
| 0x0004 | Non-Maskable Interrupt | NMI |
| 0x0008 | `Trap` Exception | Supervisor |
| 0x000C | `SingleStep` Exception | Supervisor |
| 0x0010 | `Break` Exception | Supervisor |
| 0x0014 | `UnknownInstruction` Exception | Supervisor |
| 0x0018 | Reserved | Supervisor |
| 0x001C | Reserved | Supervisor |
| 0x0020 | `DivideByZero` Exception | Supervisor |
| 0x0024 | `NullPointer` Exception | Supervisor |
| 0x0028 | `IllegalInstruction` Exception | Supervisor |
| 0x002C | `IllegalInstructionAccess` Exception | Supervisor |
| 0x0030 | `IllegalDataAccess` Exception | Supervisor |
| 0x0034 | `IllegalCodeAddress` Exception | Supervisor |
| 0x0038 | Reserved | Supervisor |
| 0x003C | Reserved | Supervisor |
| 0x0040 | Interrupt, interrupt number 0 | Interrupt |
| 0x0044 | Interrupt, interrupt number 1 | Interrupt |

| | | |
|---|---|---|
| 0x0048 | Interrupt, interrupt number 2 | Interrupt |
| ... | ... | |
| 0x0007C | Interrupt, interrupt number 15 | Interrupt |

### 2.8.7 Exceptions

The XAP4 throws exceptions to indicate that code running in User Mode has behaved improperly. The XAP4 moves into Supervisor Mode and code running in this mode can recover from the exception.

It is assumed that all Supervisor Mode code contains the defensive checks necessary to prevent exceptions occurring. The only exception condition which cannot be prevented through defensive techniques is UnknownInstruction. XAP4 provides hardware support for a reliable recovery mechanism from this exception.

#### *Trap*

The two trap instructions cause the XAP4 to move into Supervisor Mode and may be used for User Mode code to call an operating system. The argument to the trap instruction (either a register or an immediate) is passed to the `Trap` exception handler in R0_S.

#### *SingleStep*

When the T bit is set in the `FLAGS` register, this exception is triggered after every User Mode instruction, but only if that instruction has not caused any other exception (e.g., `DivideByZero`). This supports run-mode debugging and single stepping of User Mode applications.

The block instructions (`blk.*`) appear as a sequence of instructions when single stepping.

#### *Break*

The `break` exception can be caused by either a `brk` instruction or by a break condition being satisfied in the breakpoint registers.

Break events are handled according to this logic.

```
if ( (FLAGS.Mode == User) && (FLAGS.B == 1) ) then
    throw Break exception
else if debug mode enabled then
    halt
else
    nop
endif
```

Debug mode is enabled using the SIF.

When runing a program in the xIDE simulator or on an emulator, debug mode is always enabled. Any break conditions halt the processor unless the processor is in User Mode and the B flag is set.

The `brk` instructions are valid in User Mode. An operating system can prevent a User Mode task from stopping the processor by setting the B bit of the `FLAGS` register.

### *UnknownInstruction*

All opcodes that do not correspond to a valid instruction trigger this exception.

On entry to the handler, R0_S points to the undefined instruction just executed.

If an unknown instruction is executed in Supervisor Mode, the XAP4 generates an UnknownInstruction output signal which can be routed into the system's interrupt, NMI or reset circuitry. This allows a system designer to implement a highly reliable recovery mechanism for software runaways in Supervisor Mode code.

### *DivideByZero*

All forms of the divide and remainder instructions check the value of the denominator. If it is zero, then this throws a `DivideByZero` exception.

The following instructions can cause this exception:

- `div.*`
- `divrem.*`
- `rem.*`

### *NullPointer*

The XAP4 MMU detects attempts by a program to perform data accesses to memory addresses zero. For further details, refer to the XAP4 Hardware Reference Manual. Such accesses are normally the result of a program using an un-initialised pointer variable.

On entry to the handler, R6_S points to the next instruction to be executed and R0_S points to the instruction just executed.

The following instructions can cause this exception:

- `ld.i, ld.8z.i`

- ld.r, ld.8z.r
- st.i, st.8.i
- st.r, st.8.r
- stz.i, stz.8.i
- stz.r, stz.8.r
- push
- pop, pop.ret
- swap.i
- blkcp.r, blkcps.r, blkst.8.r, blkst.r

### *IllegalInstruction*

In an application using an operating system or supervisor, the XAP4 executes most code in User Mode. This mode is restricted in its access rights to certain instructions; executing any privileged instruction in User Mode triggers an `IllegalInstruction` exception.

On entry to the handler, R6_S points to the next instruction to be executed and R0_S points to the instruction just executed.

The following instructions cause this exception, because they provide access to the FLAGS register and could be used to create mode changes:

- rtie
- movr2s, movs2r
- movr2b, movb2r
- irqe, irqd

The following instructions cause this exception, because they alter the behaviour of the XAP4 core.

- sleepsif, sleepnop
- halt
- sif

The `sif` instruction is not allowed in User Mode, because it can alter the timing of accesses from external devices to the registers and memory of the processor. If SIF

cycles are required in User Mode code, the SIF instruction should be wrapped as a trap handler and made available via a trap instruction.

### IllegalInstructionAccess and IllegalDataAccess

These exceptions are triggered by the external MMU in response to the XAP4 core accessing memory that violates the current access rights (e.g., reading from memory that does not belong to the currently executing process, or writing to memory that is tagged as read-only). For further details, refer to the XAP4 Hardware Reference Manual.

`IllegalInstructionAccess` is thrown when an instruction fetch violates the access rights. `IllegalDataAccess` is thrown when an instruction's data read or write violates the access rights.

On entry to the handler R6_S points to the next instruction to be executed and R0_S points to the instruction that caused the exception. The MMU places the address that caused the exception in a memory mapped register accessible to the exception handler code.

These exceptions can be used to implement virtual memory systems or to implement memory protection schemes between different processes.

`IllegalDataAccess` can be caused by the following instructions:

- `ld.i, ld.8z.i`
- `ld.r, ld.8z.r`
- `st.i, st.8.i`
- `st.r, st.8.r`
- `stz.i, stz.8.i`
- `stz.r, stz.8.r`
- `push`
- `pop, pop.ret`
- `swap.i`
- `blkcp.r, blkcps.r, blkst.8.r, blkst.r`

### IllegalCodeAddress

All instructions are word aligned, such that `PC[0]` is zero. This exception is triggered when an odd value is written to PC, (i.e., `PC[0]` is set).

On entry to the handler, R6_S points to the next instruction to be executed and R0_S points to the instruction just executed.

The following instructions can cause this exception:

- `bra.r`
- `bsr.r`
- `pop.ret`
- `rtie`

It is not possible for the immediate forms of the branch instructions throw this exception, because they cannot encode illegal addresses.

### *IllegalStackAddress*

An `IllegalStackAddress` exception is thrown if the stack pointer is not correctly aligned when a stack operation (push or pop) is performed. The stack pointer must always be on a word boundary.

On entry to the handler, R6_S points to the next instruction to be executed and R0_S points to the instruction just executed.

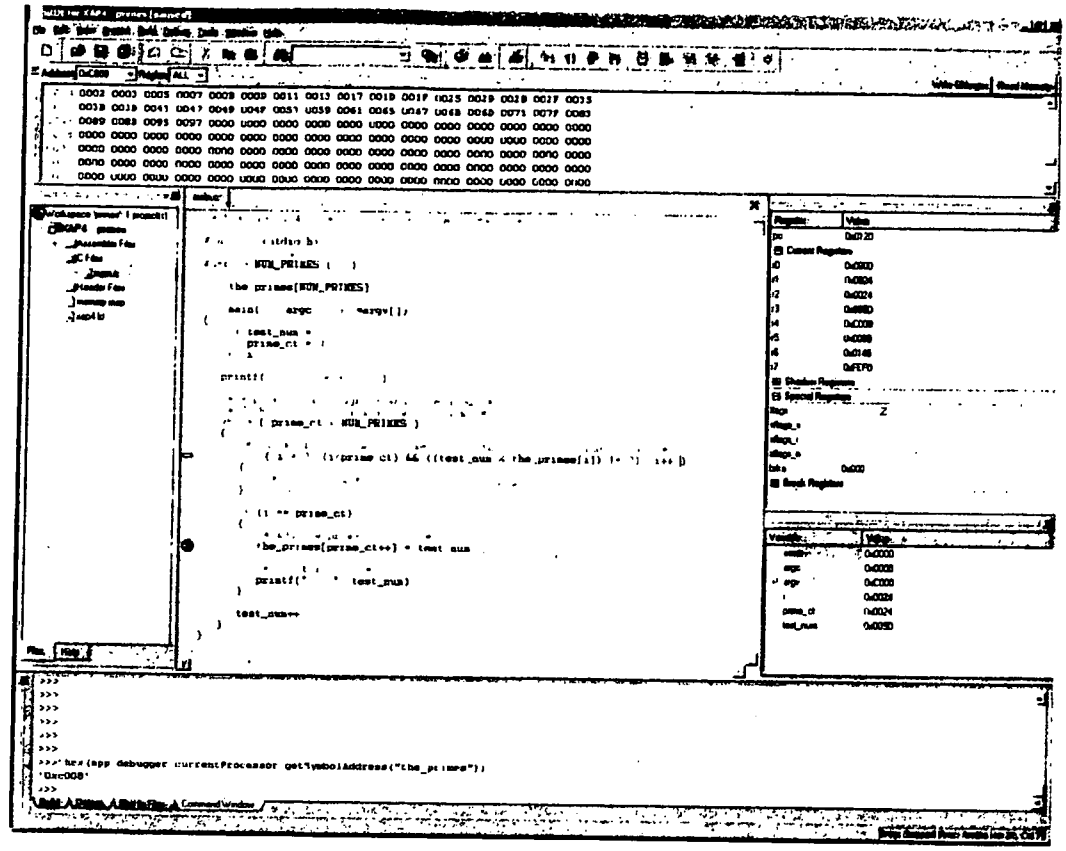
The following instructions can cause this exception:

- `push`
- `pop`
- `pop.ret`

## 2.9 Debugging

### *The xIDE Integrated Development Environment*

Systems containing a XAP4 are developed and debugged with the xIDE integrated development environment using the SIF interface.



xIDE provides:

- An integrated development and unified debug tool interface.
- Multiprocessor support.
- Familiar interface with HTML-based on-line help.
- Extensible architecture through software plug-ins.
- Project navigator to project files and program builds.
- Built-in multiple-document text editor with syntax highlighting.
- Customisable docking windows, toolbars and GUI widgets for system-specific requirements.
- Integrated Python command line. Python macros automate frequently used tasks.
- Supports Python macros to automate and simulate target system functionality.
- Multiple document windows for source code, browsing memory, debug output, register views, peripherals and variables watch.

- Support for simultaneous debug of multi-processor designs.
- Support for stop-mode and run-mode debugging.

### Stop-Mode Debugging

In stop-mode debugging, all code running on the XAP4 is stopped when a breakpoint is encountered or whilst single stepping. This is the normal debugging situation.

### Run-Mode Debugging

The XAP4 instruction set supports run-mode debugging of User Mode code. An on-chip debugger running in Supervisor Mode can use the `SingleStep` and `Break` exceptions to debug code running in User Mode.

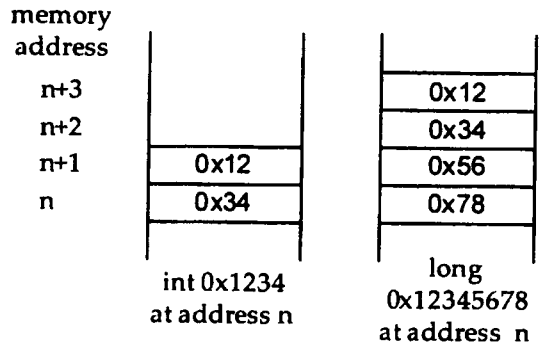
# 3 C Language Interface

## 3.1 Data Types

The XAP4 has instructions to operate on 8, 16 or 32 bit wide data. The load and store instructions transfer 8 or 16 bit wide data to and from memory.

### *Endianness*

Data and code is stored in memory in a little-endian byte order:




### *Non-native data types*

There is no hardware support for transferring 32-bit data to and from memory.

Similarly, there is no hardware support for floating point data types. The format used by the compiler is the IEEE 754 standard, implemented as little endian.

## 3.2 Data Alignment

The XAP4 supports accesses to 8-bit and 16-bit data at any memory address. Accesses to unaligned data are performed natively and with no programmer effort.

Alignment of data objects within memory is described in detail in the compiler manuals.

### 3.2.1 Aligned data

Aligned data produces the fastest code and is the normal model for the compiler. The compiler aligns data as follows:

- Single-byte variables are allocated to byte boundaries.
- All other data is allocated to a word boundary.

Fields within structures are similarly aligned. The compiler adds padding bytes between variables and within structures as necessary to achieve this alignment.

### 3.2.2 Unaligned data

A compiler option allows fields within structures to be packed. This prevents the compiler from adding padding bytes between fields and can result in fields lying at non-word-aligned addresses. Refer to the XAP4 GCC Compiler Manual for details

The XAP4 supports accesses to unaligned data natively.

Unaligned data can be accessed with the `ld.*`, `st.*`, `stz.*`, `swap.i` and `blk*` instructions.

Two memory accesses are required for each unaligned word operation. There is therefore a runtime penalty associated with unaligned data.

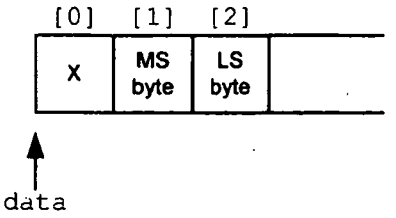
The MMU may detect accesses to unaligned data. This may provide useful debugging information, or, if speed of execution is an issue, it allows the programmer to restructure the data to remove the unaligned access. Refer to the XAP4 Hardware Reference Manual for more details.

### 3.2.3 Methods of accessing unaligned data

There are two methods of accessing unaligned data. The methods vary in their speed, readability and portability.

| | **Relative duration to access an unaligned half-word** | **Readability** | **Portability** |
|---|---|---|---|
| Read the data as a byte stream | Slow | Good | Good |
| Use C to produce unaligned accesses | Medium | Good | Poor |

The following sections illustrate both methods extracting a single little-endian 16-bit integer from an unaligned big-endian byte stream. The incoming data is in a byte buffer called data.




### *Read the data as a byte stream*

The data is read as a sequence of bytes which are reassembled into the correct order. This code is entirely portable across compilers and architectures but requires two memory reads per word extracted.

```
extern uint8 *data;
int y = (*(data+1) << 8) | (*(data+2)); // Two byte reads
```

### *Use C to produce unaligned accesses*

The first assignment generates an unaligned read from the data buffer. This code will not work on architectures that do not support unaligned memory accesses but is possible on XAP4.

```
extern uint8 *data;
int y = *(int *)(data+1);                  // Unaligned read
y = ((y&0xFF00)>>8) | ((y&0x00FF)<<8)      // Convert endianness
```

## 3.3 Calling Convention

The branch instructions (normally `bsr.p` and `bsr.a`) are used to make function calls. The effect of `bsr.p` or `bsr.a` is to transfer the address of the next instruction to the R6 (the Link Register) and to transfer the destination address to the PC. Control is returned to the calling function when the contents of R6 are transferred to PC. In practice, the compiler generates the `bsr.p` instruction.

The XAP4 calling convention is:

- On entry to a function, R6 (the Link Register) is the return address. The bsr instructions load R6 automatically. The return address must be preserved in order to return correctly to the caller.
- The called function does not need to preserve the contents of R0, R1, R2 or R6.

- The called function is responsible for preserving the remaining registers (R3 – R5, R7) across function calls. This convention is implemented in the C compiler. It is not a hardware property.
- R7 (the Stack Pointer) is used to operate a fully covered stack. All used stack is at or above the stack pointer, and the stack pointer indicates the last used location.
- Arguments are passed to functions in R0, R1 and R2, then on the stack. When an 8-bit value is passed to a function in a register, the whole register is used. The upper bits are sign or zero extended according to the type. Structures that are passed in registers are converted to a sequence of 16-bit words.
- The called function places the C function return value in R0.
- Functions may store extra return data in R1.
- Code is compiled independent of processor mode.
- When the processor is interrupted, some assembler code must be executed to store the contents of R1-R5 before the C routine can be called. This is not needed for R0 as it has a hardware shadow register.

The flags are undefined at function entry and exit.

## 3.4 Application Models

The XAP4 can support various application models.

### 3.4.1 Static linkage

This is the basic application model where the locations of all functions and data are known at link time.

### 3.4.2 Dynamic linkage

The XAP4 supports systems where applications can be loaded to memory locations that are not known at link time.

Code produced by the compiler is position independent with PC-relative addressing being used for code and constants. Load-time fixups may be necessary for certain constructs, typically involving statically allocated data.

To ensure that fixups do not change the length of an instruction, an option is available to instruct the assembler to avoid using 16-bit encodings for instructions where fixups may be necessary.

Absolute addressing can be used to access code and data at fixed addresses.

Cambridge Consultants

# 4 Instruction Set Overview

## 4.1 Summary of Assembler Syntax

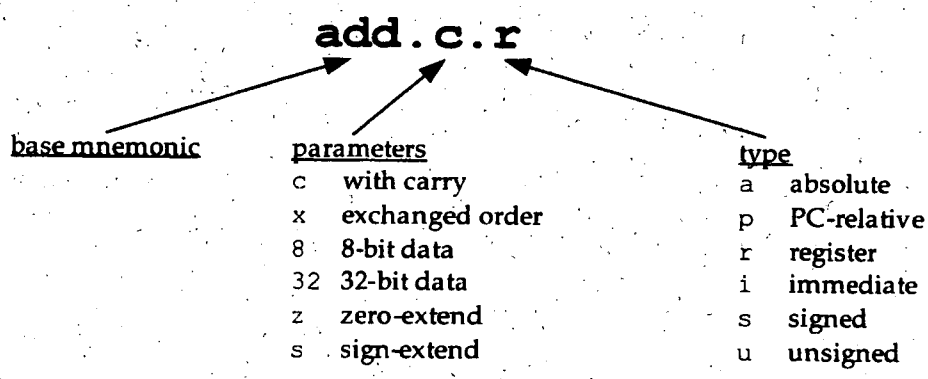
The assembler is case-sensitive. Instruction mnemonics, registers, number prefixes and directives must be in lower case.

Instructions are terminated by an end-of-line. Instructions cannot span more than one line. Spaces and tabs are treated as white space that delimits assembler tokens.

Full details of the assembler syntax are in the XAP4 GNU Binutils (Assembler, Linker and Utilities) Manual.

### 4.1.1 Instruction mnemonics

XAP4 instruction mnemonics have a regular structure, consisting of a base mnemonic, optional parameters and an optional type.




If no width is explicitly stated, the instruction operates on 16-bit data.

### 4.1.2 Operands

Operands are separated by commas.

### 4.1.3 Registers

Register names are lower case and are prefixed by %. XAP4 register names do not pollute the C namespace.

### 4.1.4 Register Lists

Some instructions take a list of registers as an operand. A RegList can take one of three forms:

- A list of one or more separate registers – `{%r3, %r6}`
- A range of registers – `{%r3-%r5}`
- Any mixture of the two forms – `{%r3-%r4, %r6}`

The registers in the RegList must be from low to high (%r0 to %r6). A register may not appear twice in the RegList. Some register combinations are invalid - refer to section 5.10, "Register Lists" for details.

### 4.1.5 Comments

C and C++ style comments are accepted:

```
/*
This is a block comment
*/
and.i %r1, %r2, #0xF00F  // This is a line comment
```

### 4.1.6 Number formats

Immediate values can be entered in decimal, hexadecimal, octal or binary. The assembler converts all immediates to 16 bits and then rejects any immediate value which cannot be represented in an instruction. Immediates outside brackets are prefixed with '#'.

#### *Decimal*

Decimal numbers require no prefix and have an optional sign. To avoid confusion with octal, the first digit must not be zero.

#### *Hexadecimal*

Hexadecimal numbers have an 0x prefix. To enter a negative number in hexadecimal, two approaches can be used:

- Explicitly define all 16 bits – for example, 0xFFFE.

- Use the unary – operator – for example, -0x2.

The characters a-f and A-F can be used in hexadecimal numbers.

### Octal

Numbers with a leading zero are treated as octal. Only digits 0 to 7 are valid in octal numbers.

### Binary

Binary numbers have an 0b prefix. To enter a negative number in binary, two approaches can be used:

- Explicitly define all 16 bits – for example, 0b1111111111111110.
- Use the unary minus operator – for example, -0b10.

### Examples of valid numbers

| Assembler syntax | Decimal equivalent |
|---|---|
| 0x1234 | 4660 |
| 0xFEDC | -292 |
| -0x1234 | -4660 |
| -0xFEDC | 292 |
| 012 | 10 |
| 0b10 | 2 |
| 0b1111111111111110 | -2 |
| -0b10 | -2 |
| -0b1111111111111110 | 2 |

## 4.1.7 Labels

Labels are case sensitive. Labels start in column 1 and end with a colon. For example:

```
// The start of the program
start:

// This is a different label
Start:
```

### 4.1.8 Directives

Directives start with a period (".") as the first non-blank character of the line. For example:

```
.org 0x1000
.file "Initialisation Code"
.text
```

Full details of the directives are in the XAP4 GNU Binutils (Assembler, Linker and Utilities) Manual.

## 4.2 Instruction Encoding

XAP4 instructions are either 32 bits or 16 bits long. Some instructions have only 32-bit encodings, some have only 16-bit encodings and some have both.

For instructions available in both 32-bit and 16-bit forms, the assmembler syntax is identical for both. The assembler normally selects the smallest encoding available.

16-bit and 32-bit instructions can be freely mixed.

An exception is `bra.p`. This instruction is commonly used to implement C switch statements. In this situation it is necessary to force the assembler to generate 16-bit or 32-bit `bra.p` instructions. There are therefore specific variants of the `bra.p` instruction:

- `bra.p.2` forces the assembler to generate a 16-bit encoding.
- `bra.p.4` forces the assembler to generate a 32-bit encoding.

## 4.3 Addressing Modes

The XAP4 has a "load/store" architecture. There are a small number of instructions which transfer memory data to and from registers. Most instructions operate on registers directly.

The only XAP4 instructions which can access memory are:

- `ld` and its variants
- `st` and its variants
- `stz` and its variants

- push and its variants
- pop and its variants
- swap
- blk and its variants

There are three addressing modes for memory operations in ld and st/stz instructions – displacement addressing, indexed addressing and PC-relative addressing.

### Displacement addressing

Displacement addressing is indicated by a .i suffix to the instruction. The instruction specifies a base address in a register and a literal offset. The memory address is calculated as the sum of the two. For example:

```
mov.i %r2, #0x1000
ld.i  %r1, @(0x20, %r2)
```

This loads R1 from memory address 0x1020.

### Indexed addressing

Indexed addressing is indicated by a .r suffix to the instruction. The instruction specifies two registers, giving a base address and an offset. The memory address is calculated as follows:

| Instruction | Calculated memory address |
|---|---|
| ld.8z.r Rd, @(Rx, Ra) | Rx + Ra |
| ld.r Rd, @(Rx, Ra) | 2*Rx + Ra |

### PC-relative addressing

PC-relative addressing is indicated by a .p suffix to the instruction. The instruction specifies a literal offset. The memory address is calculated as the sum of the literal offset and PC.

```
mov.p %r2, #0x1000 // Sets R2 = (0x1000 + PC)
ld.p  %r1, @(0x20) // Loads R1 from data at (0x20 + PC)
```

# 5 Instruction Groups

The following sections provide an overview of each group of instructions.

### 5.1.1 Branches

#### *Unconditional branches*

The XAP4 has three unconditional branches. The branch address can be an absolute address (`bra.a`), PC-relative (`bra.p`) or contained in a register (`bra.r`).

The same three operations are available when branching to a subroutine (`bsr.a`, `bsr.p` and `bsr.r`).

#### *Conditional branches*

All conditional branches are relative to the program counter. A full set of conditions is supported.

### 5.1.2 Load and Store

#### *Single memory transfers*

Loads and stores are available in 8 and 16-bit versions and with displacement, indexed and PC-relative addressing. 8-bit data is zero-extended to 16 bits on loads.

The stz instructions are used to write zeroes into memory.

#### *Swap*

The `swap.i` instruction performs an atomic swap between a register and a memory location. This is a useful instruction for implementing sempahores in operating systems. Refer to section 5.2.3, "Semaphore" for an example.

### 5.1.3 Push and Pop

The push and pop instructions enable very compact code for stack operations and function entry and exit. Multiple registers can be pushed or popped in a single instruction. The instructions are atomic.

The stack pointer is R7. The stack grows downwards in memory. Lower numbered registers are stored at lower stack addresses. The stack pointer points to the last used location on the stack.

#### *push*

The push instruction pushes a selection of registers on to the stack. The flags are not updated. An additional operand adjusts the stack pointer downwards to create a stack frame for the callee function's automatic variables.

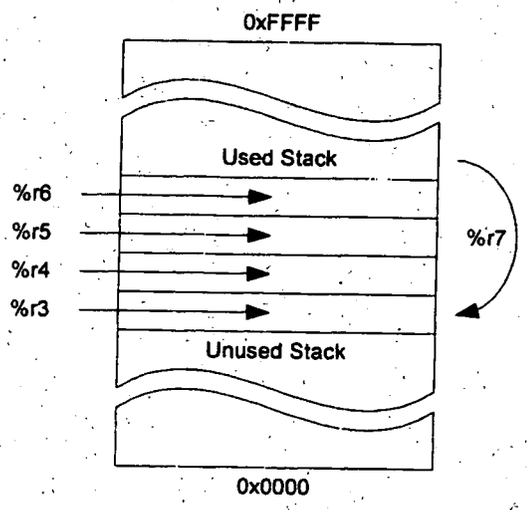
#### *push – example 1*

```
push {%r3-%r6 }, #0
```

This is equivalent to the following sequence of instructions.

```
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r6, @(0, %r7)     // Store R6 to stack memory
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r5, @(0, %r7)     // Store R5 to stack memory
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r4, @(0, %r7)     // Store R4 to stack memory
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r3, @(0, %r7)     // Store R3 to stack memory
```

This is illustrated in the diagram below.



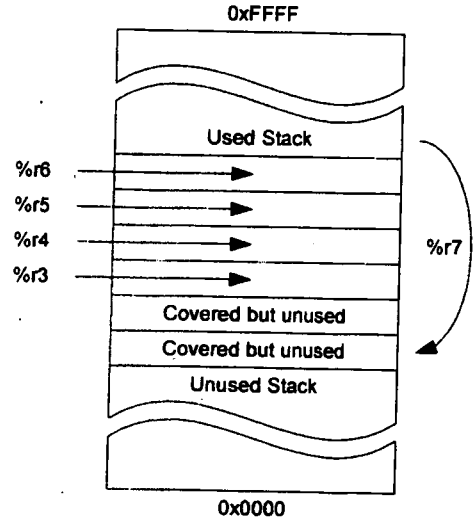

#### *push – example 2*

```
push {%r3-%r6}, #4
```

This is equivalent to the following sequence of instructions.

```
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r6, @(0, %r7)     // Store R6 to stack memory
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r5, @(0, %r7)     // Store R5 to stack memory
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r4, @(0, %r7)     // Store R4 to stack memory
add.i   %r7, %r7, #-2      // Decrease SP for 1 register
st.i    %r3, @(0, %r7)     // Store R3 to stack memory
add.i   %r7, %r7, #-4      // Decrease SP by #offset[4:1]u
                           // 4 bytes of stack free for
                           //        other purposes
```

This is illustrated in the diagram below.




## pop

The pop instruction pops a selection of registers off the stack. The flags are not updated. It is not normally necessary to pop R0, R1 or R2 from the stack as they are not preserved over a function call.

pop takes an additional operand to adjust the stack pointer upwards to remove the stack space allocated for the callee function's automatic variables.

## pop – example 1

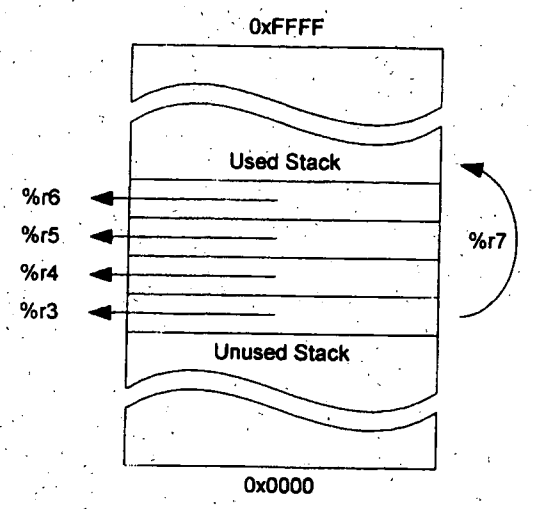
```
pop {%r3-%r6}, #0
```

This is equivalent to the following sequence of instructions:

```
ld.i    %r3, @(0, %r7)     // Load R3 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
ld.i    %r4, @(0, %r7)     // Load R4 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
ld.i    %r5, @(0, %r7)     // Load R5 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
```

```
ld.i    %r6, @(0, %r7)    // Load R6 from memory
add.i   %r7, %r7, #2      // Increase SP for 1 register
```

This is illustrated in the diagram below.



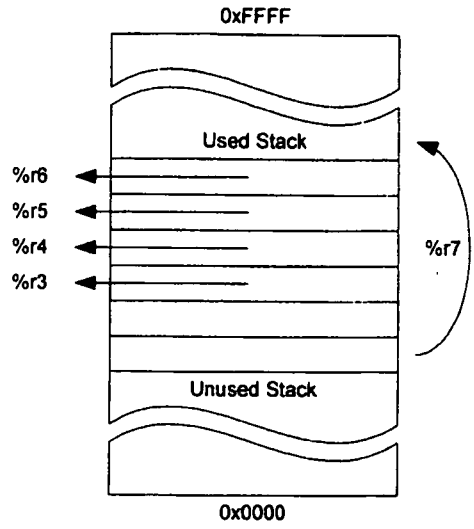

### *pop – example 2*

```
pop {%r3-%r6}, #4
```

This is equivalent to the following sequence of instructions:

```
add.i   %r7, %r7, #4      // Increase SP by #offset[4:1]u
ld.i    %r3, @(0, %r7)    // Load R3 from stack memory
add.i   %r7, %r7, #2      // Increase SP for 1 register
ld.i    %r4, @(0, %r7)    // Load R4 from stack memory
add.i   %r7, %r7, #2      // Increase SP for 1 register
ld.i    %r5, @(0, %r7)    // Load R5 from stack memory
add.i   %r7, %r7, #2      // Increase SP for 1 register
ld.i    %r6, @(0, %r7)    // Load R6 from memory
add.i   %r7, %r7, #2      // Increase SP for 1 register
```

This is illustrated in the diagram below.




### *pop.ret*

`pop.ret` performs the same operations as pop but additionally returns from the function by copying the Link Register (R6) into the program counter.

### *pop.ret – example 1*

```
pop.ret {%r3, %r5}, #0
```

This is equivalent to the following sequence of instructions:

```
ld.i    %r3, @(0, %r7)     // Load R3 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
ld.i    %r5, @(0, %r7)     // Load R5 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
bra.r   %r6                // Return to address in R6 (LR)
```

### *pop.ret example 2*

```
pop.ret {%r3, %r5}, #4
```

This is equivalent to the following sequence of instructions:

```
add.i   %r7, %r7, #4       // Increase SP by #offset[4:1]u
ld.i    %r3, @(0, %r7)     // Load R3 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
ld.i    %r5, @(0, %r7)     // Load R5 from stack memory
add.i   %r7, %r7, #2       // Increase SP for 1 register
bra.r   %r6                // Return to address in R6 (LR)
```

## 5.1.4 Move

The `mov` instructions provide various methods for loading one or more registers with literal values or other runtime constants.

#### *mov.p – PC-relative*

The instruction specifies a signed offset relative to the program counter, in the range -32KByte to +32KByte.

#### *mov.i – immediate*

A 16-bit immediate value can be loaded into a register with the `mov.i` instruction.

#### *mov.r – register to register*

`mov.r` allows a single register-to-register copy.

#### *mov.32.r – accumulator to accumulator*

`mov.32.r` allows a single accumulator-to-accumulator copy.

### 5.1.5 ALU operations

The ALU supports a diverse set of logical and arithmetic operations. The general operands are a destination register, a source register, and a third operand which may be either a third operand or an immediate value.

#### *Mutiply instructions*

XAP4 has instructions for 16x16 integer multiplies, giving either a 16-bit result or the full 32-bit result.

The `mult.sh.r` instruction produces an intermediate 32-bit result which is then shifted right a specified number of places. This is useful for fixed point multiplication.

#### *Divide and remainder instructions*

Division is performed FORTRAN-style:

- Both operands are made unsigned
- The division is performed
- The sign of the result is corrected if necessary

The div and rem instructions work together such that

```
((a/b)*b) + (a%b) = a
```

This is true for `div.s` with `rem.s` and `div.u` with `rem.u`.

The signed divide and remainder instructions truncate towards zero:

```
 5 / 2 =  2    5 % 2 =  1
-5 / 2 = -2   -5 % 2 = -1
 5 /-2 = -2    5 %-2 =  1
-5 /-2 =  2   -5 %-2 = -1
```

### *Divide by zero*

Both signed and unsigned divide, remainder and divrem instructions throw a `DivideByZero` exception if the denominator is zero and the processor is in User Mode. The result(s) of the instruction (quotient, remainder or both) are forced to zero.

A divide by zero causes the result to be forced to zero. When a divide by zero is attempted in a privileged mode, the division does not take place and the destination register remains unchanged.

The flags are updated according to whether the operation was signed or unsigned. An unsigned operation sets the C flag. A signed operation sets the V flag.

## 5.1.6 Compare operations

The compare instructions are used to set the processors's flags. These are used by the conditional branch instructions.

The 8-bit variants of the compare instructions operate on the bottom 8 bits of the operands only.

## 5.1.7 Shift and rotate

The XAP4 has signed (arithmetic) and unsigned (logical) shift right instructions. It also has shift left and rotate left instructions. Instructions can operate on either 16-bit registers or 32-bit accumulators.

A register can be shifted up to 15 places left or right in a single instruction. For left shifts, the bit shifted out is copied to the carry flag. For right shifts, the bit shifted out is copied to the carry flag.

The `rotatel` instruction performs a rotate left and does not circulate through the carry bit.

## 5.1.8 Block operations

The XAP4 has multi-cycle instructions for copying data and filling memory. These are similar to the `memcpy()`, `strncpy()` and `memset()` library functions in C.

Because the block operations can take many cycles, interrupts can occur during the execution of these instructions. All state is maintained in the registers. Unlike other

instructions, the Link Register will point to the block instruction rather than the next instruction to be executed. This allows the instruction to resume once the interrupt handler exits with `rtie`. The instruction is re-fetched after the return from interrupt.

The state of the registers when the instructions complete is:

| | | Final value after instruction | | |
|---|---|---|---|---|
| | | **blkcp.r** | **blkcps.r** | **blkst.r and blkst.8.r** |
| **Register** | Rd | Points to the destination location after the last byte copied. | | Incremented by Rt |
| | Rs | Points to the source location after the last byte copied. | | Rs is in general corrupted by `blkst.8.r`. |
| | Rt | Zero | Indicates the number of bytes not copied. This may be non-zero if a zero byte was found in the source data. | Zero |

### *Block copy – blkcp.r and blkcps.r*

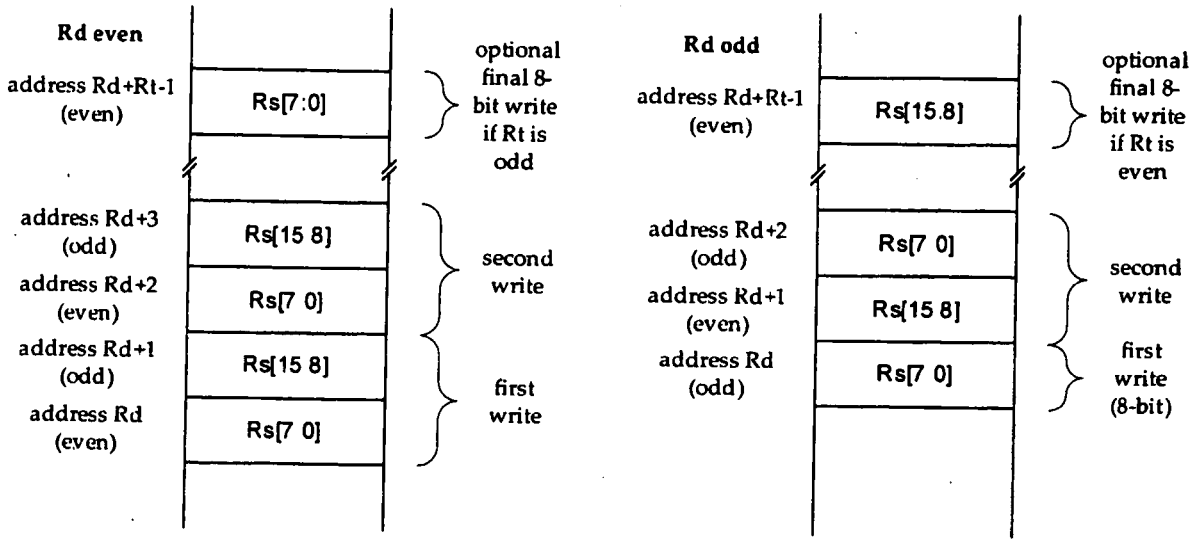
Two types of block copy are available. The `blkcp.r` instruction copies a fixed number of bytes. The `blkcps.r` instruction copies bytes up to either a maximum or a zero byte is copied.

Copies use a mixture of 8-bit and 16-bit reads and writes:

| | | Rd | |
|---|---|---|---|
| | | **Odd** | **Even** |
| **Rs** | **Odd** | An initial 8-bit operation is used, followed by 16-bit operations. An optional final 8-bit operation is used if needed. | 8-bit operations are used. |
| | **Even** | 8-bit operations are used. | 16-bit operations are used. An optional final 8-bit operation is used if needed. |

### *Block store – blkst.r*

The `blkst.r` instruction uses 16-bit writes where possible. An initial 8-bit write is used if the destination address is odd. A final 8-bit write is used if the final destination address is odd.




### *Block store – blkst.8.r*

The `blkst.8.r` instruction copies the low byte of Rs into the high byte of Rs and then uses the same write pattern as `blkst.r`. Thus, in general, `blkst.8.r` corrupts the Rs register.

## 5.1.9 Miscellaneous instructions

### *Interrupt-related*

`irqd` disables interrupts by clearing the E bit in the FLAGS register. `irqe` enables interrupts by setting the E bit. `irqs` returns the current interrupt status.

`rtie` is used to return from an interrupt or exception.

None of these instructions are permitted in User Mode.

### *Traps*

`trap.r` and `trap.i` throw a Trap exception. This allows User Mode code to call Supervisor Mode code. The trap instructions are executed as a nop in the privileged modes.

### *Sleeping and no-operation*

`sleepnop` and `sleepsif` put the XAP4 into the NOP Sleep and SIF Sleep states, respectively. The hardware WAKE_UP signal is necessary to restart the processor. In the SIF Sleep state (`sleepsif`), SIF cycles are permitted. In the NOP Sleep state (`sleepnop`), SIF cycles are not permitted. Neither instruction is permitted in User Mode.

nop is the no operation instruction.

### Debug-related

brk implements a software breakpoint. If the processor is in User Mode and the B bit in the FLAGS register is set, the Break exception is thrown. Otherwise, if the processor is in debug mode, the processor is stopped.

halt stops the processor. This instruction is not permitted in User Mode.

sif allows the processor to perform a SIF cycle. This instruction is not permitted in User Mode.

print.r causes xIDE to print a character in the debug window. This is useful for putchar emulation during simulation.

The ver and lic instructions return information about the XAP4 core.

### Operations on special registers

movr2s and movs2r provide access to the special registers.

movr2b and movb2r provide access to the breakpoint registers.

## 5.2 Common Code Sequences

### 5.2.1 Function prologue and epilogue

For a function with no stack requirement, no epilogue is necessary and the simplest function epilogue is:

```
cmp.i  %r0, #0   // Sets the flags (per calling convention)
bra.r  %r6       // Return to caller using the Link Register
```

The cmp instruction is not needed if the function returns a void.

For arbitrary functions, the prologue and epilogue code varies according to:

- Whether the function is a leaf function or not. A leaf function does not make any function calls itself.
- The amount of stack needed for the function's automatic variables, if any.

### 5.2.2 Nested interrupt prologue and epilogue

For nested interrupts, additional processor state must be saved and restored. A suitable prologue is:

```
push    {%r0-%r6}, #2          // Save R0 - R6 and allocate
                               //    one word on the stack

movs2r %r0, %sflags_i          // Get SFLAGS_I
st.i   %r0, @(0, %r7)          //    and save on the stack

irqe                           // Re-enable interrupts
```

Then on exit the original registers and flags are restored:

```
irqd                           // Disable interrupts

ld     %r0, @(0, %r7)          // Get SFLAGS_I from stack
movr2s %sflags_i, %r0          // Restore SFLAGS_I

pop    {%r0-%r6}, #2           // Restore registers R0 - R6
                               //    and free stack space

rtie                           // Return from interrupt
```

The exact register lists for the push and pop can be tailored for specific handlers to minimise stack usage and interrupt response time.

NMI handlers that are expected to return control to the interrupted process may use similar entry and exit code, with the omission of the `irqe` and `irqd` instructions

### 5.2.3 Semaphore

The `swap.i` instruction can be used to implement a semaphore without needing to disable interrupts. Two assembly functions are needed – one to lock (acquire) a semaphore, and a second to release it.

`osXSimpleGetSem()` attempts to acquire a semaphore. The `swap.i` instruction swaps-in the "locked" value, and gives the current semaphore state in R3.

```
   .text

   .globl osXSimpleGetSem, osXSimpleReleaseSem

// osXSimpleGetSem
// ===============
// Description
//   Try to acquire the semaphore specified in R1.  Returns immediately
//   if the semaphore is already locked.
// Entry:
//   R1 = semaphore address
// Exit:
```

```
//   R1 = 0 indicates the semaphore was acquired successfully
//   R1 = 1 indicates another task already has the semaphore
//
osXSimpleGetSem:
   mov.i  %r2, #1
   swap.i %r2, @(0, %r1)
   mov.r  %r1, %r2
   cmp.r  %r1, #0
   bra.r  %r6
```

osXSimpleReleaseSem() uses the swap.i instruction to set the semaphore state to "unlocked". The value retrieved from the semaphore should be 1 but this simple code does not check that the semaphore was locked on entry; nor that it was locked by the current process.

```
   .text

   .globl osXSimpleReleaseSem

// osXSimpleReleaseSem
// ===================
// Description
//   Releases a semaphore.  No checking is performed (eg, that this
//   task acquired the semaphore successfully in the first place).
// Entry:
//   R1 = semaphore address
// Exit:
//   R1 = undefined
//
osXSimpleReleaseSem:
   mov.i  %r2, #0
   swap.i %r2, @(0, %r1)
   bra.r  %r6
```

These functions can be used from C as follows:

```
extern int  osXSimpleGetSem    (int *);
extern void osXSimpleReleaseSem(int *);

int mySemaphore;

{
      if ( 0 == osXSimpleGetSem(mySemaphore) )
      {
          // We have the semaphore
          ...
          osXSimpleReleaseSem(mySemaphore);
      }
}
```

### 5.2.4 Initialisation of User Mode registers

Code running in Supervisor Mode can initialise the User Mode registers before starting a User Mode task. R0, R6 (the Link Register) and R7 (the Stack Pointer) need

explicit initialisation. Registers R2-R5 and the FLAGS register are shared with Supervisor Mode and can be initialised with normal instructions.

```
osXInitUserRegs:
   // Initialise R0, R6 (LR) and R7 (SP)
   mov.i  %r0, 0x1111        // Initial value for user R0
   movr2s %r1, %r0_u         // Load user R0

   mov.p  %r1, #killfunction // To kill the user task if it ends
   movr2s %r1, %r6_u         // Load user R6 (Link Register)

   mov.i  %r1, #stacktop     // Top of user stack
   movr2s %r1, %r7_u         // Load user R7 (Stack Pointer)

   // Return to caller
   bra.r  %r6
```

## 5.3 Instructions Grouped by Function

### 5.3.1 Branches

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Unconditional Branch | bra.a | Branch, absolute | ---- |
| | bra.p<br>bra.p.2<br>bra.p.4 | Branch, PC-relative | ---- |
| | bra.r | Branch, register | ---- |
| Unconditional Branch Subroutine | bsr.a | Branch to subroutine , absolute | ---- |
| | bsr.p | Branch to subroutine, PC-relative | ---- |
| | bsr.r | Branch to subroutine, register | ---- |
| Conditional Branch | bcc | Branch if carry clear | ---- |
| | bcs | Branch if carry set | ---- |
| | beq | Branch if equal | ---- |
| | bge.s | Branch if greater than or equal, signed | ---- |
| | bge.u | Branch if greater than or equal, unsigned | ---- |
| | bgt.s | Branch if greater than, signed | ---- |
| | bgt.u | Branch if greater than, unsigned | ---- |
| | ble.s | Branch if less than or equal, signed | ---- |
| | ble.u | Branch if less than or equal, unsigned | ---- |
| | blt.s | Branch if less than, signed | ---- |
| | blt.u | Branch if less than, unsigned | ---- |

Cambridge Consultants

| | | | |
|---|---|---|---|
| | bmi | Branch if minus | ---- |
| | bne | Branch if not equal | ---- |
| | bpl | Branch if plus | ---- |
| | bvc | Branch if overflow clear | ---- |
| | bvs | Branch if overflow set | ---- |

### 5.3.2 Load and Store

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Load (displacement) | ld.8z.i | Load, 8-bit, zero-extend, displacement | ZN-- |
| | ld.i | Load, displacement | ZN-- |
| Load (indexed) | ld.8z.r | Load, 8-bit, zero-extend, indexed | ZN-- |
| | ld.r | Load, indexed | ZN-- |
| Load (PC-relative) | ld.8z.p | Load, 8-bit, zero-extend, PC-relative | ZN-- |
| | ld.p | Load, PC-relative | ZN-- |
| Store (displacement) | st.8.i | Store, 8-bit, displacement | ---- |
| | st.i | Store, displacement | ---- |
| | stz.8.i | Store zero, 8-bit, displacement | ---- |
| | stz.i | Store zero, displacement | ---- |
| Store (indexed) | st.8.r | Store, 8-bit, indexed | ---- |
| | st.r | Store, indexed | ---- |
| | stz.8.r | Store zero, 8-bit, indexed | ---- |
| | stz.r | Store zero, indexed | ---- |
| Store (PC-relative) | st.8.p | Store, 8-bit, PC-relative | ---- |
| | st.p | Store, PC-relative | ---- |
| | stz.8.p | Store zero, 8-bit, PC-relative | ---- |
| | stz.p | Store zero, PC-relative | ---- |
| Swap | swap.i | Swap register with memory | ZN-- |

### 5.3.3 Push and Pop

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Push | push | Push to stack | ---- |
| Pop | pop | Pop from stack | ---- |
| | pop.ret | Pop from stack and return | ---- |

### 5.3.4 Move

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Move | mov.i | Move, immediate | ZN-- |
| | mov.p | Move, PC-relative | ---- |
| | mov.r | Move, register | ZN-- |
| | mov.32.r | Move, accumulator | ZN-- |

### 5.3.5 ALU operations

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Add | add.c.i | Add with carry, immediate | ZNCV |
| | add.c.r | Add with carry, register | ZNCV |
| | add.i | Add, immediate | ZNCV |
| | add.r | Add, register | ZNCV |
| Subtract | sub.c.r | Subtract, with carry, register | ZNCV |
| | sub.r | Subtract, register | ZNCV |
| | sub.x.i | Subtract, exchanged order,immediate | ZNCV |
| | sub.xc.i | Subtract, with carry, exchanged order, immediate | ZNCV |
| Logical | and.i | And, immediate | ZN-- |
| | and.r | And, register | ZN-- |
| | or.i | Or, immediate | ZN-- |
| | or.r | Or, register | ZN-- |
| | xor.i | Exclusive-or, immediate | ZN-- |
| | xor.r | Exclusive-or, register | ZN-- |
| Multiply | mult.32s.i | Multiply, 32-bit signed, immediate | ZN-- |
| | mult.32s.r | Multiply, 32-bit signed, register | ZN-- |
| | mult.32u.i | Multiply, 32-bit unsigned, immediate | ZN-- |
| | mult.32u.r | Multiply, 32-bit unsigned, register | ZN-- |
| | mult.i | Multiply, immediate | ZN-- |
| | mult.sh.r | Multiply, signed shift right, register | ZN-V |
| | mult.r | Multiply, register | ZN-- |
| Divide and Remainder | div.32s.i | Divide, 32-bit signed, immediate | ZN-V |
| | div.32s.r | Divide, 32-bit signed, register | ZN-V |
| | div.32u.i | Divide, 32-bit unsigned, immediate | ZNC- |

| | | | |
|---|---|---|---|
| | div.32u.r | Divide, 32-bit unsigned, register | ZNC- |
| | div.s.i | Divide, signed, immediate | ZN-V |
| | div.s.r | Divide, signed, register | ZN-V |
| | div.u.i | Divide, unsigned, immediate | ZNC- |
| | div.u.r | Divide, unsigned, register | ZNC- |
| | divrem.32s.i | Divide and remainder, 32-bit signed, immediate | ---V |
| | divrem.32s.r | Divide and remainder, 32-bit signed, register | ---V |
| | divrem.32u.i | Divide and remainder, 32-bit unsigned, immediate | --C- |
| | divrem.32u.r | Divide and remainder, 32-bit unsigned, register | --C- |
| | divrem.s.i | Divide and remainder, signed, immediate | ---V |
| | divrem.s.r | Divide and remainder, signed, register | ---V |
| | divrem.u.i | Divide and remainder, unsigned, immediate | --C- |
| | divrem.u.r | Divide and remainder, signed, register | --C- |
| | rem.32s.i | Remainder, 32-bit signed, immediate | ZN-V |
| | rem.32s.r | Remainder, 32-bit signed, register | ZN-V |
| | rem.32u.i | Remainder, 32-bit unsigned, immediate | ZNC- |
| | rem.32u.r | Remainder, 32-bit unsigned, register | ZNC- |
| | rem.s.i | Remainder, signed, immediate | ZN-V |
| | rem.s.r | Remainder, signed, register | ZN-V |
| | rem.u.i | Remainder, unsigned, immediate | ZNC- |
| | rem.u.r | Remainder, unsigned, register | ZNC- |

### 5.3.6 Compare operations

| | 16-bit | 8-bit |
|---|---|---|
| Immediate | cmp.i | cmp.8.i |
| Register | cmp.r | cmp.8.r |
| Immediate with carry | cmp.c.i | cmp.8c.i |
| Register with carry | cmp.c.r | cmp.8c.r |
| Immediate, exchanged order | cmp.x.i | cmp.8x.i |
| Immediate with carry, exchanged order | cmp.xc.i | cmp.8xc.i |

All cmp instructions affect the ZNCV flags.

### 5.3.7 Shift and rotate

| Function | Mnemonic | Description | Flags |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Shift left | shiftl.32.i | Shift left, 32-bit, immediate | ZNC- |
| | shiftl.32.r | Shift left, 32-bit, register | ZNC- |
| | shiftl.i | Shift left, immediate | ZNC- |
| | shiftl.r | Shift left, register | ZNC- |
| Shift right | shiftr.32s.i | Shift right, 32-bit signed, immediate | ZNC- |
| | shiftr.32s.r | Shift right, 32-bit signed, register | ZNC- |
| | shiftr.32u.i | Shift right, 32-bit unsigned, immediate | ZNC- |
| | shiftr.32u.r | Shift right, 32-bit unsigned, register | ZNC- |
| | shiftr.s.i | Shift right, signed, immediate | ZNC- |
| | shiftr.s.r | Shift right, signed, register | ZNC- |
| | shiftr.u.i | Shift right, unsigned, immediate | ZNC- |
| | shiftr.u.r | Shift right, unsigned, register | ZNC- |
| Rotate | rotatel.32.i | Rotate left, 32-bit, immediate | ZNC- |
| | rotatel.32.r | Rotate left, 32-bit, register | ZNC- |
| | rotatel.i | Rotate left, immediate | ZNC- |
| | rotatel.r | Rotate left, register | ZNC- |

### 5.3.8 Block copy and store

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Block copy | blkcp.r | Block copy, register | ---- |
| | blkcps.r | Block copy string, register | ---- |
| Block store | blkst.8.r | Block store, 8-bit, register | ---- |
| | blkst.r | Block store, register | ---- |

### 5.3.9 Miscellaneous instructions

| Function | Mnemonic | Description | Flags |
|---|---|---|---|
| Sign extend | sext.r | Sign-extend, register | ZN-- |
| System instructions | brk | Break | ---- |
| | halt | Halt | ---- |
| | nop | No operation | ---- |
| | print.r | Print, register | ---- |
| | sleepnop | Sleep | ---- |
| | sleepsif | Sleep and allow SIF | ---- |

| | | | |
|---|---|---|---|
| Debug | lic | Read XAP4 licence number | ---- |
| | movb2r | Move breakpoint to register | ZN-- |
| | movr2b | Move register to breakpoint | ---- |
| | sif | SIF | ---- |
| | ver | Read XAP4 version number | ---- |
| Interrupt and exceptions | irqd | Interrupt disable | ---- |
| | irqe | Interrupt enable | ---- |
| | irqs | Interrupt status | ZN-- |
| | rtie | Return from interrupt or exception | ZNCV |
| | trap.i | Trap, immediate | ZNCV |
| | trap.r | Trap, register | ZNCV |
| Special registers | movs2r | Move special to register | ZN-- |
| | movr2s | Move register to special | ---- |

## 5.4 Alphabetical List of All Instructions

The XAP4 instruction set is listed alphabetically below. The "Sizes" column indicates whether the instruction can be encoded in 16 bits, 32 bits or both.

| Mnemonic | Operands | Operation | Sizes |
|---|---|---|---|
| add.c.i | Rd, Rs, #immediate | Add with carry, immediate | 32 |
| add.c.r | Rd, Rs, Rt | Add with carry, register | 16 |
| add.i | Rd, Rs, #immediate | Add, immediate | 16, 32 |
| add.r | Rd, Rs, Rt | Add, register | 16 |
| and.i | Rd, Rs, #immediate | And, immediate | 16, 32 |
| and.r | Rd, Rs, Rt | And, register | 16 |
| bcc | label | Branch if carry clear | 32 |
| bcs | label | Branch if carry set | 32 |
| beq | label | Branch if equal | 16, 32 |
| bge.s | label | Branch if greater than or equal, signed | 16, 32 |
| bge.u | label | Branch if greater than or equal, unsigned | 32 |
| bgt.s | label | Branch if greater than, signed | 16, 32 |

| Mnemonic | Operands | Operation | Sizes |
|---|---|---|---|
| bgt.u | label | Branch if greater than, unsigned | 16, 32 |
| ble.s | label | Branch if less than or equal, signed | 16, 32 |
| ble.u | label | Branch if less than, unsigned. | 16, 32 |
| blkcp.r | Rd, Rs, Rt | Block copy, register | 32 |
| blkcps.r | Rd, Rs, Rt | Block copy string, register | 32 |
| blkst.8.r | Rd, Rs, Rt | Block store, 8-bit, register | 32 |
| blkst.r | Rd, Rs, Rt | Block store, register | 32 |
| blt.s | label | Branch if less than, signed | 16, 32 |
| blt.u | label | Branch if less than, unsigned | 32 |
| bmi | label | Branch if minus | 32 |
| bne | label | Branch if not equal | 16, 32 |
| bpl | offset | Branch if plus | 32 |
| bra.a | address | Branch, absolute | 32 |
| bra.p<br>bra.p.2<br>bra.p.4 | label | Branch, PC-relative | 16, 32 |
| bra.r | Ra | Branch, register | 16 |
| brk | | Break | 16 |
| bsr.a | address | Branch to subroutine, absolute | 32 |
| bsr.p | label | Branch to subroutine, PC-relative | 32 |
| bsr.r | Ra | Branch to subroutine, register | 16 |
| bvc | label | Branch if overflow clear | 32 |
| bvs | label | Branch if overflow set | 32 |
| cmp.8.i | Rs, #immediate | Compare, immediate | 32 |
| cmp.8.r | Rs, Rt | Compare, register | 16 |
| cmp.8c.i | Rs, #immediate | Compare, 8-bit with carry, immediate | 32 |
| cmp.8c.r | Rs, Rt | Compare, 8-bit with carry, register | 16 |
| cmp.8x.i | Rs, #immediate | Compare, 8-bit, exchanged order, immediate | 32 |
| cmp.8xc.i | Rs, #immediate | Compare, 8-bit with carry, exchanged order, immediate | 32 |

Cambridge Consultants

| Mnemonic | Operands | Operation | Sizes |
|---|---|---|---|
| cmp.c.i | Rs, #immediate | Compare, with carry, immediate | 32 |
| cmp.c.r | Rs, Rt | Compare, with carry, register | 16 |
| cmp.i | Rs, #immediate | Compare, immediate | 16, 32 |
| cmp.r | Rs, Rt | Compare, register | 16 |
| cmp.x.i | Rs, #immediate | Compare, exchanged order, immediate | 32 |
| cmp.xc.i | Rs, #immediate | Compare, with carry, exchanged order, immediate | 32 |
| div.32s.i | Rd, Rs, #immediate | Divide, 32-bit signed, immediate | 32 |
| div.32s.r | Rd, Rs, Rt | Divide, 32-bit signed, register | 32 |
| div.32u.i | Rd, Rs, #immediate | Divide, 32-bit unsigned, immediate | 32 |
| div.32u.r | Rd, Rs, Rt | Divide, 32-bit unsigned, register | 32 |
| div.s.i | Rd, Rs, #immediate | Divide, signed, immediate | 32 |
| div.s.r | Rd, Rs, Rt | Divide, signed, register | 32 |
| div.u.i | Rd, Rs, #immediate | Divide, unsigned, immediate | 32 |
| div.u.r | Rd, Rs, Rt | Divide, unsigned, register | 32 |
| divrem.32s.i | Rd, Rs, #immediate | Divide & remainder, 32-bit signed, immediate | 32 |
| divrem.32s.r | Rd, Rs, Rt | Divide & remainder, 32-bit signed, register | 32 |
| divrem.32u.i | Rd, Rs, #immediate | Divide & remainder, 32-bit unsigned, immediate | 32 |
| divrem.32u.r | Rd, Rs, Rt | Divide & remainder, 32-bit unsigned, register | 32 |
| divrem.s.i | Rd, Rs, #immediate | Divide & remainder, signed, immediate | 32 |
| divrem.s.r | Rd, Rs, Rt | Divide & remainder, signed, register | 32 |
| divrem.u.i | Rd, Rs, #immediate | Divide & remainder, unsigned, immediate | 32 |
| divrem.u.r | Rd, Rs, Rt | Divide & remainder, unsigned, register | 32 |
| halt | | Halt | 16 |
| irqd | | Disable interrupts | 16 |
| irqe | | Enable interrupts | 16 |
| irqs | Rd | Interrupt status | 16 |

Cambridge Consultants

| Mnemonic | Operands | Operation | Sizes |
|---|---|---|---|
| ld.8z.i | Rd, @(offset, Ra) | Load, 8-bit, zero-extend, displacement | 16, 32 |
| ld.8z.p | Rd, @(offset) | Load, 8-bit, zero-extend, PC-relative | 32 |
| ld.8z.r | Rd, @(Rx, Ra) | Load, 8-bit, zero-extend, indexed | 16 |
| ld.i | Rd, @(offset, Ra) | Load, displacement | 16, 32 |
| ld.p | Rd, @(offset) | Load, PC-relative | 32 |
| ld.r | Rd, @(Rx, Ra) | Load, indexed | 16 |
| lic | Rd | Read XAP4 licence number | 16 |
| mov.32.r | Rd, Rs | Move, accumulator | 16 |
| mov.i | Rd, #immediate | Move, immediate | 16, 32 |
| mov.p | Rd, #label | Move, PC-relative | 32 |
| mov.r | Rd, Rs | Move, register | 16 |
| movb2r | Rd, BRs | Move breakpoint to register | 32 |
| movr2b | BRd, Rs | Move register to breakpoint | 32 |
| movr2s | SRd, Rs | Move register to special | 32 |
| movs2r | Rd, SRs | Move special to register | 32 |
| mult.32s.i | Rd, Rs, #immediate | Multiply, 32-bit signed, immediate | 32 |
| mult.32s.r | Rd, Rs, Rt | Multiply, 32-bit signed, register | 32 |
| mult.32u.i | Rd, Rs, #immediate | Multiply, 32-bit unsigned, immediate | 32 |
| mult.32u.r | Rd, Rs, Rt | Multiply, 32-bit unsigned, register | 32 |
| mult.i | Rd, Rs, #immediate | Multiply immediate | 32 |
| mult.r | Rd, Rs, Rt | Multiply, register | 16 |
| mult.sh.r | Rd, Rs, Rt, #immediate | Multiply, signed shift right, register | 32 |
| nop | | No operation | 16 |
| or.i | Rd, Rs, #immediate | Or, immediate | 32 |
| or.r | Rd, Rs, Rt | Or, register | 16 |
| pop | RegList, #offset | Pop from stack | 16 |
| pop.ret | RegList, #offset | Pop from stack and return | 16 |
| print.r | Rs | Print, register | 16 |

Cambridge Consultants

| Mnemonic | Operands | Operation | Sizes |
|---|---|---|---|
| push | ReqList, #offset | Push to stack | 16 |
| rem.32s.i | Rd, Rs, #immediate | Remainder, 32-bit signed, immediate | 32 |
| rem.32s.r | Rd, Rs, Rt | Remainder, 32-bit signed, register | 32 |
| rem.32u.i | Rd, Rs, #immediate | Remainder, 32-bit unsigned, immediate | 32 |
| rem.32u.r | Rd, Rs, Rt | Remainder, 32-bit unsigned, register | 32 |
| rem.s.i | Rd, Rs, #immediate | Remainder, signed, immediate | 32 |
| rem.s.r | Rd, Rs, Rt | Remainder, signed, register | 32 |
| rem.u.i | Rd, Rs, #immediate | Remainder, unsigned, immediate | 32 |
| rem.u.r | Rd, Rs, Rt | Remainder, unsigned, register | 32 |
| rotatel.32.i | Rd, Rs, #immediate | Rotate left, 32-bit, immediate | 32 |
| rotatel.32.r | Rd, Rs, Rt | Rotate left, 32-bit, register | 32 |
| rotatel.i | Rd, Rs, #immediate | Rotate left, immediate | 16 |
| rotatel.r | Rd, Rs, Rt | Rotate left, register | 16 |
| rtie | | Return from interrupt/exception | 16 |
| sext.r | Rd | Sign extend, register | 16 |
| shiftl.32.i | Rd, Rs, #immediate | Shift left, 32-bit, immediate | 32 |
| shiftl.32.r | Rd, Rs, Rt | Shift left, 32-bit, register | 32 |
| shiftl.i | Rd, Rs, #immediate | Shift left, immediate | 16 |
| shiftl.r | Rd, Rs, Rt | Shift left, register | 16 |
| shiftr.32s.i | Rd, Rs, #immediate | Shift right, 32-bit signed, immediate | 32 |
| shiftr.32s.r | Rd, Rs, Rt | Shift right, 32-bit signed, register | 32 |
| shiftr.32u.i | Rd, Rs, #immediate | Shift right, 32-bit unsigned, immediate | 32 |
| shiftr.32u.r | Rd, Rs, Rt | Shift right, 32-bit unsigned, register | 32 |
| shiftr.s.i | Rd, Rs, #immediate | Shift right, signed, immediate | 16 |
| shiftr.s.r | Rd, Rs, Rt | Shift right, signed, register | 16 |
| shiftr.u.i | Rd, Rs, #immediate | Shift right, unsigned, immediate | 16 |
| shiftr.u.r | Rd, Rs, Rt | Shift right, unsigned, register | 16 |
| sif | | SIF | 16, 32 |
| sleepnop | | Sleep | 16 |

| Mnemonic | Operands | Operation | Sizes |
|---|---|---|---|
| sleepsif | | Sleep and allow SIF | 16 |
| st.8.i | Rs, @(offset, Ra) | Store, 8-bit, displacement | 16, 32 |
| st.8.p | Rs, @(offset) | Store, 8-bit, PC-relative | 32 |
| st.8.r | Rs, @(Rx, Ra) | Store, 8-bit, indexed | 16 |
| st.i | Rs, @(offset, Ra) | Store, displacement | 16, 32 |
| st.p | Rs, @(offset) | Store, displacement | 32 |
| st.r | Rs, @(Rx, Ra) | Store, indexed | 16 |
| stz.8.i | @(offset, Ra) | Store zero, 8-bit, displacement | 16, 32 |
| stz.8.p | @(offset) | Store zero, 8-bit, PC-relative | 32 |
| stz.8.r | @(Rx, Ra) | Store zero, 8-bit, indexed | 32 |
| stz.i | @(offset, Ra) | Store zero, displacement | 16, 32 |
| stz.p | @(offset) | Store zero, PC-relative | 32 |
| stz.r | @(Rx, Ra) | Store zero, indexed | 32 |
| sub.c.r | Rd, Rs, Rt | Subtract, with carry, register | 16 |
| sub.r | Rd, Rs, Rt | Subtract, register | 16 |
| sub.x.i | Rd, Rs, #immediate | Subtract, exchanged order, immediate | 32 |
| sub.xc.i | Rd, Rs, #immediate | Subtract, exchanged order, with carry, immediate | 32 |
| swap.i | Rd, @(0, Ra) | Swap register with memory | 32 |
| trap.i | #immediate | Trap, immediate | 32 |
| trap.r | Rs | Trap, register | 32 |
| ver | Rd | Read XAP4 version number | 16 |
| xor.i | Rd, Rs, #immediate | Exclusive-or, immediate | 32 |
| xor.r | Rd, Rs, Rt | Exclusive-or, register | 16 |

## 5.5 Privileged Instructions

The following instructions are not permitted in User Mode:

rtie, movr2s, movs2r, movr2b, movb2r, irqe, irqd, sleepsif, sleepnop, halt, sif

These instructions either provide access to the FLAGS register (thereby allowing them to change the processor mode), or otherwise are capable of affecting the operation of the XAP4 core.

If a privileged instruction is executed in User Mode, the XAP4 throws an `IllegalInstruction` exception.

## 5.6 Aliased Instructions

The assembler translates a small number of instructions into other, equivalent instuctions. These are:

| Original instruction | Translated into |
|---|---|
| `blt.u label` | `bcs label` |
| `bge.u label` | `bcc label` |

The instructions are disassembled into the translated instructions by xIDE.

## 5.7 Register Naming in Instructions

Register operands are named as follows:

| Register symbol | Description |
|---|---|
| Rd | Destination register. |
| Rs | Primary source register. |
| Rt | Secondary source register. |
| Ra | Register containing a base address. This is interpreted as a byte address. |
| Rx | Register containing an index. This is interpreted according to the width of the instruction. Rx is used in the indexed addressing mode versions of the load and store instructions. |

## 5.8 Register Specification Fields

### *3-bit register fields*

Registers are generally specified with a 3-bit field. The mapping from the 3-bit field to the 16-bit register set is as follows:

| 3-bit R field | register |
|---|---|
| 000 | %r0 |
| 001 | %r1 |
| 010 | %r2 |
| 011 | %r3 |
| 100 | %r4 |
| 101 | %r5 |
| 110 | %r6 |
| 111 | %r7 |

### 2-bit register fields

In some instructions, registers are specified with a 2-bit field. This is mapped to the register set as follows:

| 2-bit R field | register |
|---|---|
| 00 | %r0 |
| 01 | %r1 |
| 10 | %r2 |
| 11 | %r7 |

### Special register fields

The mapping from a 3-bit special register field to the special registers is:

| SRx | register |
|---|---|
| 000 | `%flags` |
| 001 | `%sflags_s` |
| 010 | `%sflags_i` |
| 011 | `%sflags_n` |
| 100 | `%brke` |
| 101 | `%r0_u` |
| 110 | `%r6_u` |
| 111 | `%r7_u` |

### Breakpoint register fields

The mapping from a 3-bit breakpoint register field to the breakpoint register is:

| BRx | register |
|---|---|
| 000 | %brk0 |
| 001 to 111 | Invalid |

## 5.9 Immediates

Many instructions take an immediate operand which is encoded in the instruction. For a subset of the permitted immediate values, the instruction encoding rules allow 16-bit immediates to be compressed in the instruction encoding. The rules governing immediate encoding are:

- Zero or sign extension.
- implicit bits.

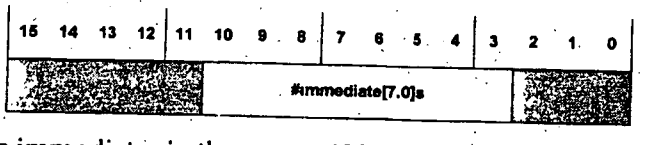
### *Zero or sign extension*

In the 16-bit instruction encoding, the immediate field is typically 4 or 8 bits wide. In some of these instructions, the XAP4 extends the immediate to 16 bits before use. This is either a zero-extension or a sign-extension, depending on the instruction.

Sign-extension of an immediate is indicated by an "s" suffix in the immediate field in the instruction descriptions. Zero-extension is indicated by a "u" suffix.

If the assembler cannot represent the immediate in the shortened form, it selects the equivalent 32-bit encoding of the instruction.

An example of sign-extension is the 16-bit encoding of the `add.i` instruction:

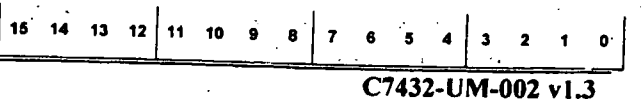



This encoding can represent positive immediates in the range -128 to +127. Immediates outside this range require the 32-bit encoding of the instruction which contains a full 16-bit immediate.

### *Implicit bits*

In some cases, not all bits of the operand are encoded in the instruction.

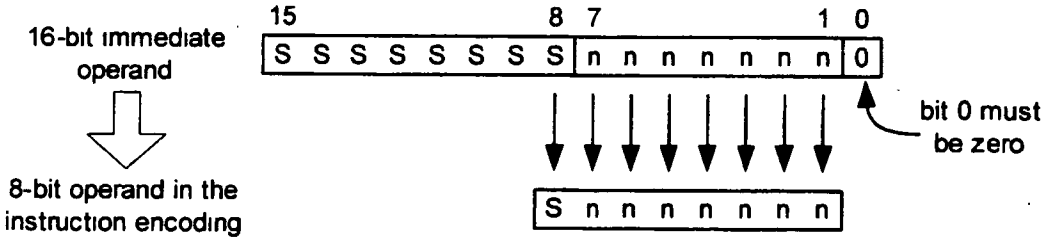
An example of implicit bits is the 16-bit encoding of the `beq` instruction.






This syntax indicates that only bits 8 to 1 of the immediate are encoded in the instruction. Bit 0 is an implicit 0. The assembler rejects any `beq` instructions which have an odd immediate value.

The "s" suffix indicates that bits 15 to 9 are sign-extended from bit 8.




This encoding can represent immediates in the range -256, -254, +252, +254. Immediates outside this range require the 32-bit encoding of the instruction which contains a full 16-bit immediate.

## 5.10 Register List Fields

The four-bit RegList fields are used in the `push`, `pop` and `pop.ret` instructions and are interpreted as follows:

| 4-bit RegList | register(s) |
|---|---|
| 0000 | {%r0} |
| 0001 | {%r1} |
| 0010 | {%r2} |
| 0011 | {%r3} |
| 0100 | {%r4} |
| 0101 | {%r5} |
| 0110 | {%r6} |
| 0111 | {} |
| 1000 | {%r3, %r6} |
| 1001 | {%r3, %r4} |
| 1010 | {%r3, %r4, %r6} |
| 1011 | {%r3, %r4, %r5} |
| 1100 | {%r3, %r4, %r5, %r6} |

| | |
|---|---|
| 1101 | {%r1, %r2} |
| 1110 | {%r0, %r1, %r2} |
| 1111 | {%r0, %r1, %r2, %r3, %r4, %r5, %r6} |

# 6 Instruction Set Reference

The following pages describe each instruction in detail. Instructions are ordered alphabetically.

# add.c.i — Add with carry, immediate

| | | |
|---|---|---|
| **Instruction** | | `add.c.i Rd, Rs, #immediate` |
| **Description** | | Add immediate and the carry bit to a register |
| **Flags** | **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| | **N** | Set if the result is negative; cleared if the result is positive |
| | **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| | **V** | Set if the result of the signed operation is not correct; cleared otherwise |
| **Operation** | | `Rd = Rs + #immediate + FLAGS.C` |
| **Usage Notes** | | `add.c.i` can be used for signed or unsigned, integer or fixed-point arithmetic.<br>The Z flag behaviour is useful for 32-bit arithmetic. |
| **Examples** | | `add.c.i %r1, %r2, #10` |

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | Rd | Rs | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Cambridge Consultants

# add.c.r — Add with carry, register

**Instruction** `add.c.r Rd, Rs, Rt`

**Description** Add two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| **V** | Set if the result of the signed operation is not correct; cleared otherwise |

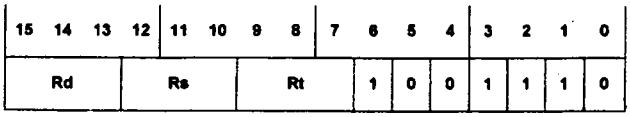
**Operation** `Rd = Rs + Rt + FLAGS.C`

**Usage Notes** `add.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

**Examples** `add.c.r %r1, %r2, %r3`

**16-bit Encoding**



# add.i — Add, immediate

**Instruction** `add.i Rd, Rs, #immediate`

**Description** Add a register and an immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| **V** | Set if the result of the signed operation is not correct; cleared otherwise |

**Operation** `Rd = Rs + #immediate`

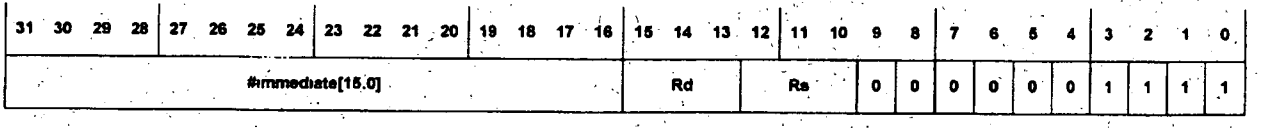
**Usage Notes** `add.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

For small immediates, a 16-bit encoding of this instruction is available.

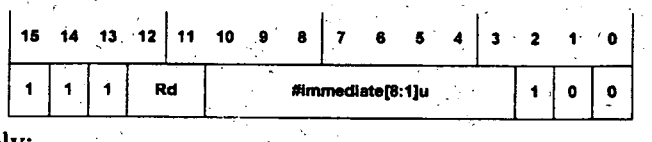
**Examples**

```
add.i %r1, %r2, #0x1234
add.i %r1, %r2, #0xFEDC
```

### 32-bit Encoding



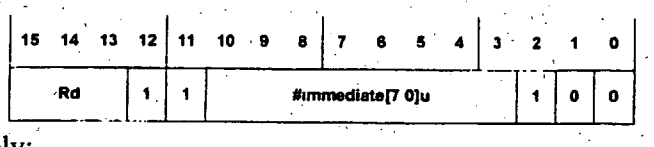

### 16-bit Encoding - 1




The following rules apply:

- Rs is %r7.
- The 2-bit Rd field can specify %r0, %r1, %r2 or %r7.
- This encoding can represent immediates in the range 0, 2, ..., 508, 510.

### 16-bit Encoding - 2

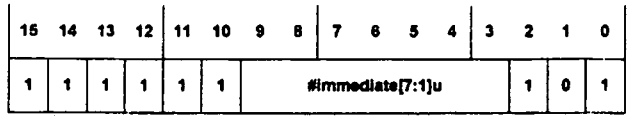



The following rules apply:

- The 3-bit Rd field specifies both Rd and Rs (%r0 to %r6).
- This encoding can represent immediates in the range 0, 1, ..., 254,

255.

**16-bit Encoding - 3**

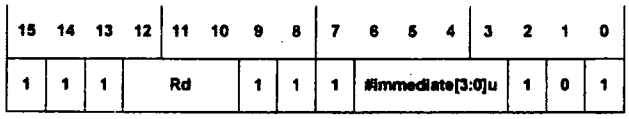


| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | #immediate[7:1]u | | | | | | | 1 | 0 | 1 |

The following rules apply:

- Rd and Rs are %r7.
- The immediate value is (#immediate[7:1]u – 256) giving the range -2, -4, ..., -254, -256.

**16-bit Encoding - 4**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rd | | | 1 | 1 | 1 | #immediate[3:0]u | | | | 1 | 0 | 1 |

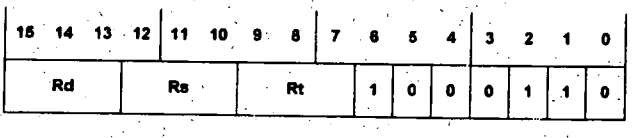
The following rules apply:

- The 3-bit Rd field specifies both Rd and Rs (%r0 to %r6).
- The immediate value is (#immediate[3:0]u – 16) giving the range -1, -2, ..., -15, -16.

# add.r — Add, register

| | | |
|---|---|---|
| **Instruction** | | `add.r Rd, Rs, Rt` |
| **Description** | | Add two registers |
| **Flags** | **Z** | Set if the result is zero; cleared if the result is non-zero |
| | **N** | Set if the result is negative; cleared if the result is positive |
| | **C** | Set if the result of the unsigned operation is not correct; cleared otherwise |
| | **V** | Set if the result of the signed operation is not correct; cleared otherwise |
| **Operation** | | `Rd = Rs + Rt` |
| **Usage Notes** | | `add.r` can be used for signed or unsigned, integer or fixed-point arithmetic. |
| **Examples** | | `add.r %r1, %r2, %r3` |

**16-bit Encoding**



# and.i — And, immediate

**Instruction** `and.i Rd, Rs, #immediate`

**Description** Logical AND of register with an immediate value

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared if the result is non-zero |
| **N** | Set if the result is negative; cleared if the result is positive |
| **C** | Unchanged |
| **V** | Unchanged |

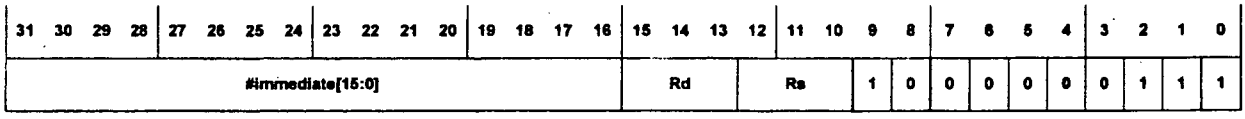
**Operation** `Rd = Rs & #immediate`

**Usage Notes** A number of commonly used combinations of registers and immediates are available in two 16-bit encodings of this instruction.

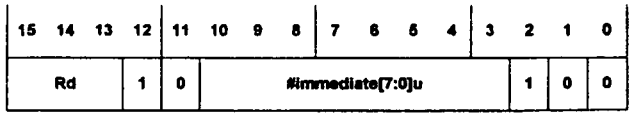
**Examples**

```
and.i %r2, %r1, #0x1234
and.i %r2, %r1, #0xFEDC
```

### 32-bit Encoding




### 16-bit Encoding

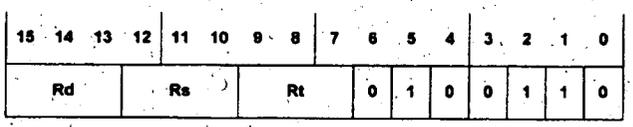



The following rules apply:

- In this encoding, Rd is the same as Rs and is in the range %r0 to %r6.
- This encoding can represent immediates in the range 0 to 255.

# and.r — And, register

| | | |
|---|---|---|
| **Instruction** | | `and.r Rd, Rs, Rt` |
| **Description** | | Logical AND of two registers |
| **Flags** | **Z** | Set if the result is zero; cleared if the result is non-zero |
| | **N** | Set if the result is negative; cleared if the result is positive |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs & Rt` |
| **Usage Notes** | | Rd must be in the range %r0 to %r6. |
| **Examples** | | `and.r %r1, %r2, %r3` |
| **16-bit Encoding** | | |



Cambridge Consultants

# bcc — Branch if carry clear

**Instruction** `bcc label`

**Description** Branch if carry clear

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.C == 0 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bcc label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

Cambridge Consultants

# bcs — Branch if carry set

**Instruction** `bcs label`

**Description** Branch if carry set

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.C == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bcs label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

Cambridge Consultants

# beq — Branch if equal

**Instruction** `beq label`

**Description** Branch if equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.Z == 1 ) then
   PC = PC + offset
endif
```

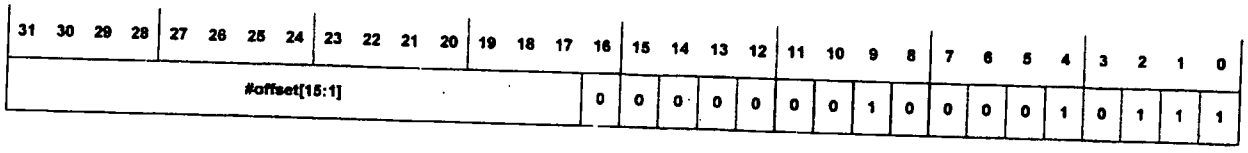
**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

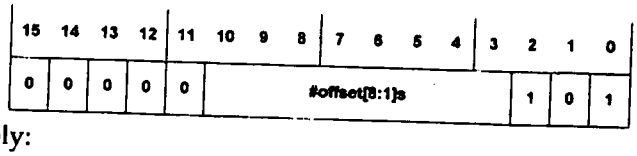
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `beq label`

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# bge.s — Branch if greater than or equal, signed

**Instruction** `bge.s label`

**Description** Branch if greater than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

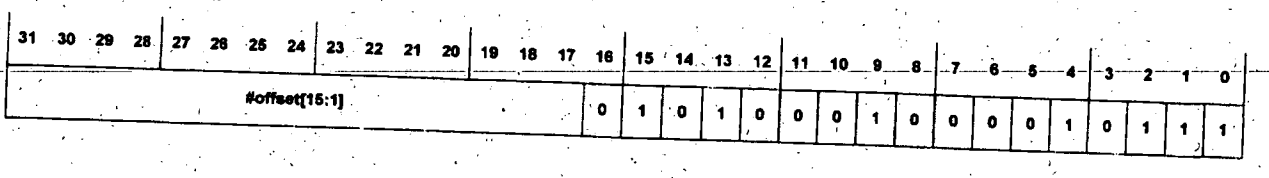
```
PC = next instruction
if ( FLAGS.N == FLAGS.V ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

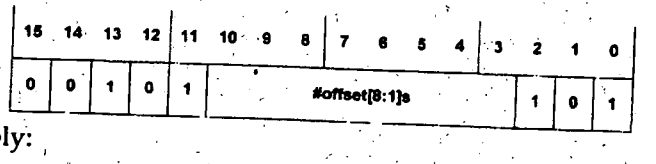
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `bge.s label`

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# bge.u — Branch if greater than or equal, unsigned

**Instruction** `bge.u label`

**Description** Branch if greater than

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.C == 0 ) then
   PC = PC + offset
endif
```

**Usage Notes**

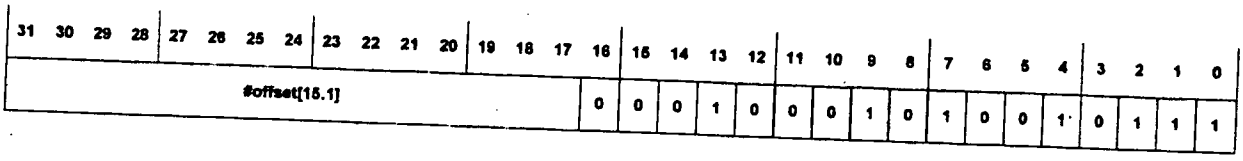
The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

This instruction is translated into `bcc` by the assember. xIDE prints this instruction as `bcc label`.

**Examples**

```
bge.u label
```

**32-bit Encoding**



# bgt.s — Branch if greater than, signed

**Instruction** `bgt.s label`

**Description** Branch if greater than, signed

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( ( FLAGS.N == FLAGS.V ) && ( FLAGS.Z == 0 ) ) then
   PC = PC + offset
endif
```

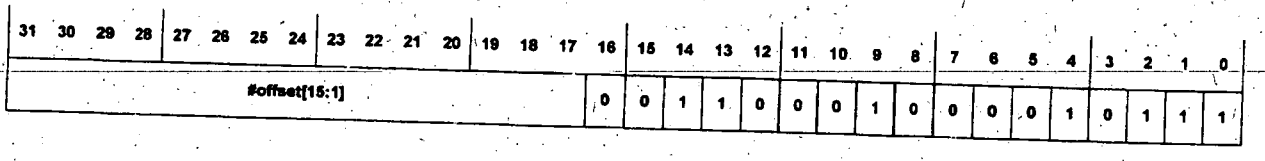
**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

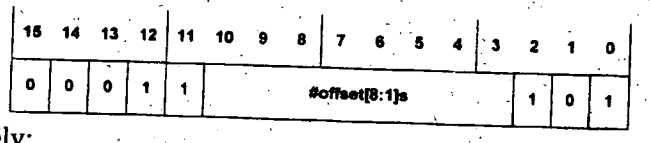
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `bgt.s label`

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# bgt.u — Branch if greater than, unsigned

**Instruction** `bgt.u label`

**Description** Branch if greater than, unsigned

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

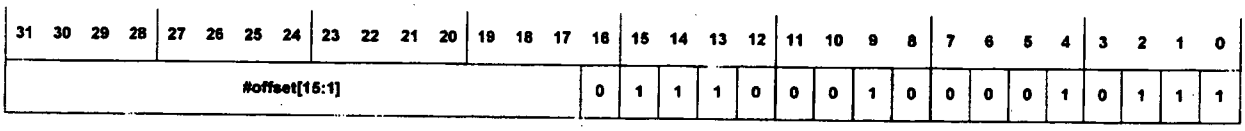
```
PC = next instruction
if ( ( FLAGS.C == 0 ) && ( FLAGS.Z == 0 ) ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

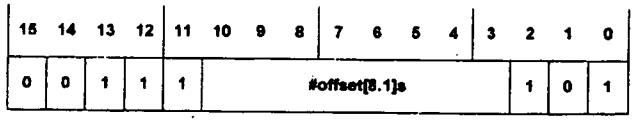
**Examples** `bgt.u label`

### 32-bit Encoding



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #offset[16:1] | | | 0 | 1 1 1 0 | 0 0 1 0 | 0 0 0 1 | 0 1 1 1 |

### 16-bit Encoding



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| 0 0 1 1 | 1 | #offset[8.1]s | 1 0 1 |

The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# ble.s — Branch if less than or equal, signed

**Instruction** `ble.s label`

**Description** Branch if less than or equal, signed

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( ( FLAGS.N != FLAGS.V ) || ( FLAGS.Z == 1 ) ) then
   PC = PC + offset
endif
```

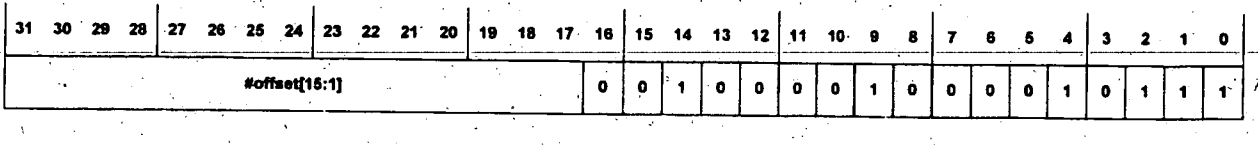
**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

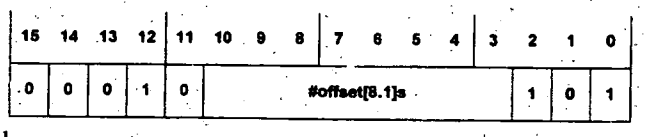
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `ble.s label`

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# ble.u — Branch if less than or equal, unsigned

**Instruction** `ble.u label`

**Description** Branch if less than or equal, unsigned

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

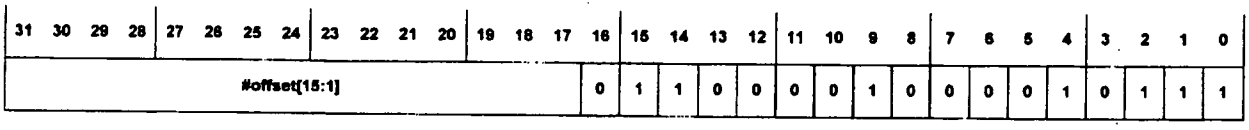
```
PC = next instruction
if ( ( FLAGS.C == 1 ) || ( FLAGS.Z == 1 ) ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

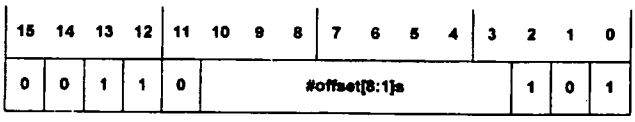
**Examples** `ble.u label`

### 32-bit Encoding



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | 0 | 1 1 0 0 | 0 0 1 0 | 0 0 0 1 | 0 1 1 1 |

### 16-bit Encoding



| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| 0 0 1 1 | 0 | #offset[8:1]s | 1 0 1 |

The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# blkcp.r — Block copy, register

**Instruction** `blkcp.r Rd, Rs, Rt`

**Description** Copy Rt bytes from Rs to Rd

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
while (Rt > 0)
{
    Rt--
    *Rd++ = *Rs++
}
```

**Usage Notes**

Rt bytes of data are copied from the address specified in Rs to the address specified in Rd. The source and destination addresses increment during the copy. If the source area and destination area overlap, part of the source data will be overwritten.

The instruction can be interrupted before the copy is complete. When the interrupt handler exits, the copy will resume.

This instruction is useful for implementing `memcpy()`-like functions.

Refer to section 5.1.8, "Block operations" for more details of this instruction.

If Rt is zero, the instruction copies no data.

A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
blkcp.r %r1, %r0, %r3
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd | Rs | Rt | 0 0 0 | 0 0 1 0 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 |

# blkcps.r — Block copy string, register

**Instruction** `blkcps.r Rd, Rs, Rt`

**Description** Copy Rt bytes from Rs to Rd

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
while (Rt > 0)
{
    Rt--
    if ( (*Rd++ = *Rs++ ) == '\0' ) break
}
```

**Usage Notes**

Up to Rt bytes of data are copied from the address specified in Rs to the address specified in Rd. The copy stops on the first of:

- The count reaching zero.
- The copied byte being zero (a '\0' character).

The final '\0' byte is copied.

The source and destination addresses increment during the copy. If the source area and destination area overlap, part of the source data will be overwritten.

The instruction can be interrupted before the copy is complete. When the interrupt handler exits, the copy will resume.

This instruction is useful for implementing `strcpy()` and `strncpy()`-like functions.

Refer to section 5.1.8, "Block operations" for more details of this instruction.

If Rt is zero, the instruction copies no data. If Rt is 0xFFFF, the instruction copies a maximum of 65535 bytes of memory. This can be used to implement `strcpy()`, where the copy is terminated only by a '\0' character.

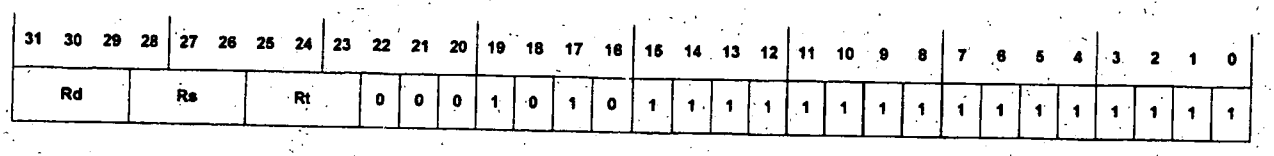
A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
blkcps.r %r1, %r0, %r3
```

Cambridge Consultants

### 32-bit Encoding



# blkst.8.r — Block store, 8-bit, register

**Instruction** `blkst.8.r Rd, Rs, Rt`

**Description** Stores a register to a block of memory

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
while (Rt > 0)
{
    Rt--
    *Rd++ = Rs
}
```

**Usage Notes**

The low 8 bits of Rs are stored to a block of memory starting at an address specified in Rd and length specified in Rt.

The destination address increments during the store.

The instruction can be interrupted before the store is complete. When the interrupt handler exits, the store resumes.

This instruction is useful for implementing `memset()`-like functions.

Refer to section 5.1.8, "Block operations" for more details of this instruction.

If Rt is zero, the instruction stores no data.

A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `blkst.8.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# blkst.r — Block store, register

**Instruction** `blkst.r Rd, Rs, Rt`

**Description** Stores a register to a block of memory

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
while (Rt > 0)
{
    Rt--
    *Rd++ = Rs
}
```

**Usage Notes**

Rs is stored to a block of memory starting at an address specified in Rd and length specified in Rt.

The destination address increments during the store.

The instruction can be interrupted before the store is complete. When the interrupt handler exits, the store resumes.

This instruction is useful for implementing `memset()`-like functions.

Refer to section 5.1.8, "Block operations" for more details of this instruction.

If Rt is zero, the instruction stores no data.

To fill an area of memory with zeros, use %r0 for Rs.

A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `blkst.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd | Rs | Rt | 0 0 1 | 0 0 1 0 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 |

# blt.s — Branch if less than, signed

**Instruction** `blt.s label`

**Description** Branch if less than, signed

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.N != FLAGS.V ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `blt.s label`

### 32-bit Encoding

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

### 16-bit Encoding

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | #offset[8.1]s | | | | | | | 1 | 0 | 1 | |

The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# blt.u — Branch if less than, unsigned

**Instruction** `blt.u offset`

**Description** Branch if less than

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.C == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

This instruction is translated into `bcs` by the assember. xIDE prints this instruction as `bcs label`.

**Examples**

```
blt.u label
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

# bmi — Branch if minus

**Instruction** `bmi label`

**Description** Branch if minus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.N == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bmi label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

# bne — Branch if not equal

**Instruction** `bne label`

**Description** Branch if not equal

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.Z == 0 ) then
  PC = PC + offset
endif
```

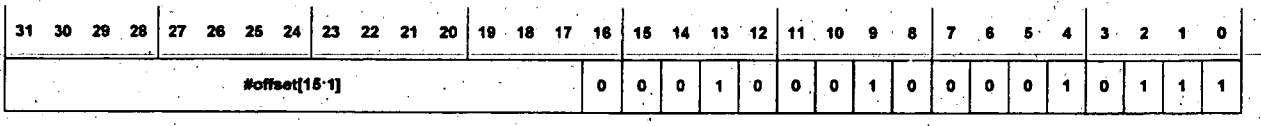
**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

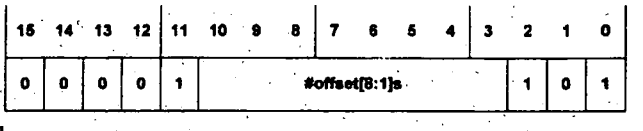
**Examples** `bne label`

**32-bit Encoding**



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

**16-bit Encoding**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | #offset[8:1]s | | | | | | | | 1 | 0 | 1 |

The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# bpl — Branch if plus

**Instruction** `bpl label`

**Description** Branch if plus

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.N == 0 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bpl label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15 1] | | | | | | | | | | | | | | | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

# bra.a — Branch, absolute

**Instruction** `bra.a address`

**Description** Unconditional branch to an absolute address

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `PC = address`

**Usage Notes** This instruction can branch to any address in the 64KByte of memory.

All instructions start on a 16-bit boundary, so the least significant bit of the address must be zero.

**Examples** `bra.a label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #address[15.1] | | | | | | | | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

Cambridge Consultants

# bra.p, bra.p.2, bra.p.4 — Branch, PC-relative

**Instruction**

```
bra.p   label
bra.p.2 label
bra.p.4 label
```

**Description** Unconditional branch relative to the program counter

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = label
```

**Usage Notes**

The assembler inserts the offset from the current PC to the label into the instruction. All instructions start on a 16-bit boundary, so the least significant bit of the offset must be zero.

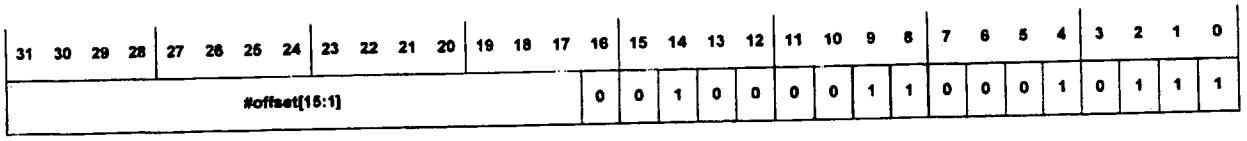
The 16-bit encoding can encode offsets in the range -256 to +256.

The assembler selects the 16-bit encoding for `bra.p` if the offset is in the range -256 to +256. `bra.p.2` forces the assembler to use the 16-bit encoding. `bra.p.4` forces the assembler to use the 32-bit encoding.

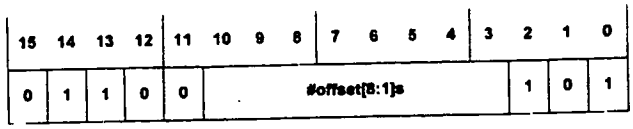
**Examples**

```
bra.p   label
bra.p.2 label
bra.p.4 label
```

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- This encoding can represent offsets in the range -256, -254, ..., +252, +254.

# bra.r — Branch, register

**Instruction** `bra.r Ra`

**Description** Unconditional branch to the address specified in the register Ra

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `PC = Ra`

**Usage Notes** This instruction can be used to return from simple subroutines (normally `bra.r %r6`).

This instruction throws an `IllegalCodeAddress` exception if bit 0 of Rs is not zero.

**Examples** `bra.r %r6`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Ra | | | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

# brk — Break

**Instruction** `brk`

**Description** Throw a break exception or halt the processor

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
if (FLAGS.Mode == User && FLAGS.B == 1) then
      throw Break exception
else if debug mode enabled then
      halt
else
      nop
endif
```

**Usage Notes**

The `brk` instruction is used for debugging.

If the processor is in User Mode and the B bit of the `FLAGS` register is set, the `Break` exception is thrown. This allows an on-chip debugger or operating system to deal with the break.

Otherwise, if the processor is in debug mode the processor is stopped.

When debugging with the XAP4 simulator, the processor is always in debug mode. On the XAP4 emulator and real XAP4s, debug mode is controlled with the SIF interface.

The `FLAGS` register can be written with the `movr2s` instruction. This instruction is not permitted in User Mode.

The processor can be restarted using the SIF interface or a reset.

**Examples**

```
brk
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# bsr.a — Branch to subroutine, absolute

**Instruction** `bsr.a address`

**Description** Unconditional branch to subroutine at an absolute address

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC  = next instruction
R6  = PC
PC  = address
```

**Usage Notes** The `bsr.a` instruction changes the program counter and saves the address of the instruction after the branch into the Link Register, R6.

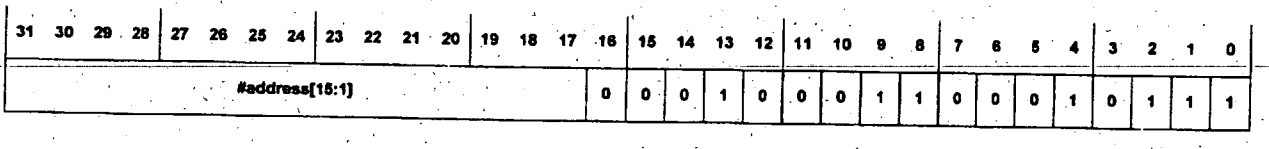
The address can be in the range 0 to 64KByte.

All instructions start on a 16-bit boundary, so the least significant bit of the address must be zero.

**Examples**

```
bsr.a label
```

**32-bit Encoding**



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #address[15:1] | | | | | | | | | | | | | | | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

# bsr.p — Branch to subroutine, PC-relative

**Instruction**

```
bsr.p label
```

**Description**

Unconditional branch to a subroutine at an address relative to the current PC

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC  = next instruction
R6  = PC
PC  = label
```

**Usage Notes**

The `bsr.p` instruction changes the program counter and saves the address of the instruction after the branch into the Link Register, R6.

The assembler inserts the offset from the current PC to the label into the instruction. All instructions start on a 16-bit boundary, so the least significant bit of the offset must be zero.

The offset can take values in the range -32KByte to +32KByte.

**Examples**

```
bsr.p label
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[16.1] | | | | | | | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

# bsr.r — Branch to subroutine, register

**Instruction** `bsr.r Ra`

**Description** Unconditional branch to a subroutine at an address contained in Ra

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC  = next instruction
R6  = PC
PC  = Ra
```

**Usage Notes**

The `bsr.r` instruction changes the program counter and saves the address of the instruction after the branch into the Link Register, R6.

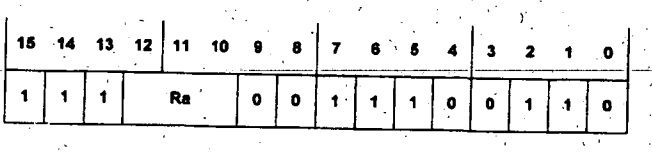
Ra cannot be %r6.

This instruction throws an `IllegalCodeAddress` exception if bit 0 of Rs is not zero.

**Examples**

```
bsr.r %r1
```

**16-bit Encoding**



# bvc — Branch if overflow clear

**Instruction** `bvc label`

**Description** Branch if overflow clear

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
PC = next instruction
if ( FLAGS.V == 0 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples** `bvc label`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:1] | | | | | | | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

# bvs — Branch if overflow set

**Instruction** `bvs label`

**Description** Branch if overflow set

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

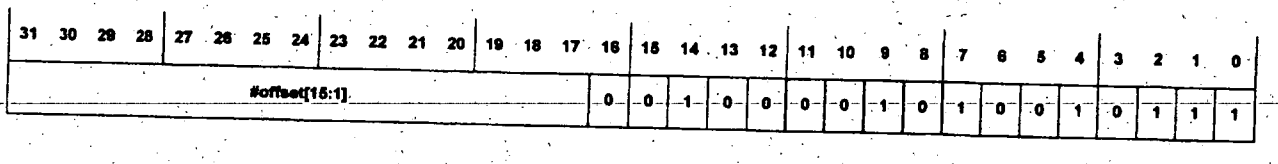
```
PC = next instruction
if ( FLAGS.V == 1 ) then
   PC = PC + offset
endif
```

**Usage Notes** The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32KByte to +32KByte. All instructions start on a 16-bit boundary, so the least significant bit of the offset is always zero.

**Examples**

```
bvs label
```

**32-bit Encoding**



# cmp.8.i — Compare, 8-bit, immediate

**Instruction** `cmp.8.i Rs, #immediate`

**Description** An 8-bit compare of the register with an immediate

**Flags**

- **Z** Set if `Rs[7:0] == #immediate[7:0]`; cleared otherwise
- **N** Set if the result of the subtraction is negative; cleared if the result is positive
- **C** Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
- **V** Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

**Operation** `Rs[7:0] - #immediate[7:0]`

**Usage Notes** This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register. The high 8 bits of the immediate and register are ignored.

The immediate can take values in the range -128 to +255.

**Examples** `cmp.8.i %r1, #0x12`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[7:0] | | | | | | | | 0 | 0 | 0 | Rs | | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | Rs | | | 1 | 1 | 1 | #immediate[3:0]u | | | 1 | 0 | 1 | |

The following rules apply:

- The 3-bit Rs field can specify %r0 to %r6.
- This encoding can represent immediates in the range 0, 1, ...,14, 15.

# cmp.8.r — Compare, 8-bit, register

**Instruction**

```
cmp.8.r Rs, Rt
```

**Description**

Compare the low 8 bits in two registers

**Flags**

**Z** Set if `Rs[7:0] == Rt[7:0]`; cleared otherwise

**N** Set if the result of the subtraction is negative; cleared if the result is positive

**C** Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise

**V** Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

**Operation**

```
Rs[7:0] - Rt[7:0]
```

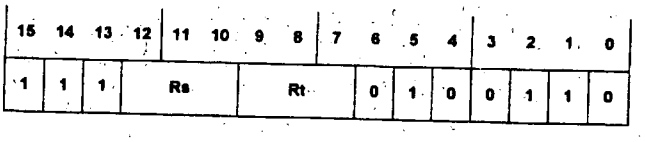
**Usage Notes**

This is an 8-bit compare of the lower 8 bits from two registers. The high 8 bits of the two registers are ignored.

**Examples**

```
cmp.8.r %r1, %r2
```

**16-bit Encoding**



# cmp.8c.i Compare, 8-bit with carry, immediate

**Instruction** `cmp.8c.i Rs, #immediate`

**Description** An 8-bit compare of the register with an immediate and the carry flag

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation** `Rs[7:0] - #immediate[7:0] - FLAGS.C`

**Usage Notes** This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 8 bits of the immediate and register are ignored.

The immediate can take values in the range -128 to +255.

**Examples** `cmp.8c.i %r1, #0x12`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | #immediate[7:0] | | | | | | | | 0 | 1 | 0 | Rs | | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

## cmp.8c.r — Compare, 8-bit with carry, register

**Instruction**

```
cmp.8c.r Rs, Rt
```

**Description**

An 8-bit compare of two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation**

```
Rs[7:0] - Rt[7:0] - FLAGS.C
```

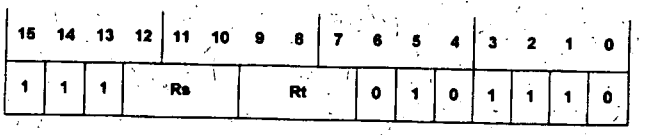
**Usage Notes**

This is an 8-bit compare of the lower 8 bits from two registers and the carry bit. The high 8 bits of the two registers are ignored.

**Examples**

```
cmp.8c.r %r1, %r2
```

**16-bit Encoding**



# cmp.8x.i — Compare, 8-bit, exchange, immediate

**Instruction**

```
cmp.8x.i Rs, #immediate
```

**Description**

An 8-bit compare of the register with an immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs[7:0] == #immediate[7:0]`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation**

```
#immediate[7:0] - Rs[7:0]
```

**Usage Notes**

This is an 8-bit compare of the lower 8 bits of a register with an 8-bit immediate. The high 8 bits of the register and immediate are ignored.

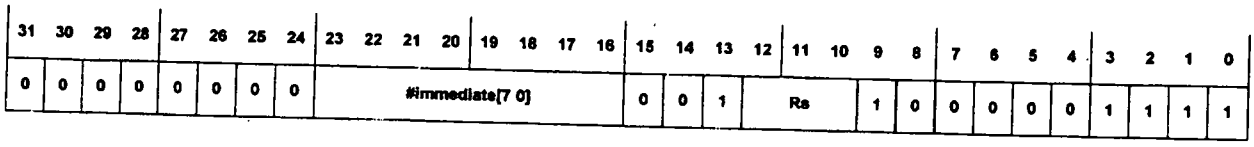
The immediate can take values in the range -128 to +255.

This is the same as `cmp.8.i` but with the operand order reversed.

**Examples**

```
cmp.8x.i %r1, #0x12
```

**32-bit Encoding**



# cmp.8xc.i Compare, 8-bit carry, exchange, immediate

**Instruction**

```
cmp.8xc.i Rs, #immediate
```

**Description**

An 8-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise |

**Operation**

```
#immediate[7:0] - Rs[7:0] - FLAGS.C
```

**Usage Notes**

This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 8 bits of the immediate and register are ignored.

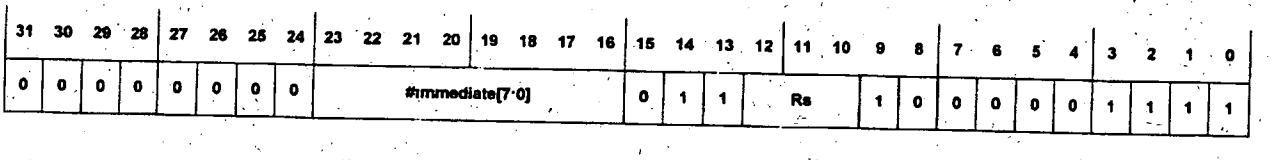
The immediate can take values in the range -128 to +255.

This is the same as `cmp.8c.i` but with the operand order reversed.

**Examples**

```
cmp.8xc.i %r1, #0x12
```

**32-bit Encoding**



# cmp.c.i — Compare, with carry, immediate

**Instruction**

```
cmp.c.i Rs, #immediate
```

**Description**

A 16-bit compare of the register with an immediate and the carry flag

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation**

```
Rs - #immediate - FLAGS.C
```

**Usage Notes**

This is a 16-bit compare of an immediate with a register and the carry bit.

The Z flag behaviour is useful for 32-bit arithmetic.

A 32-bit compare of accumulator %r1 with an immediate (for example, 0x12345678) can be performed in two instructions:

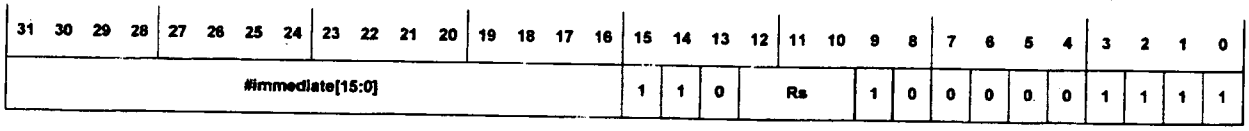
```
cmp.i   %r1, 0x5678
cmp.c.i %r1, 0x1234
```

**Examples**

```
cmp.c.i %r1, #0x1234

cmp.c.i %r1, #0xFEDC
```

**32-bit Encoding**



## cmp.c.r — Compare, with carry, register

**Instruction** `cmp.c.r Rs, Rt`

**Description** A 16-bit compare of two registers and the carry bit

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs - Rt - FLAGS.C`

**Usage Notes** This is a 16-bit compare of two registers and the carry bit.

The Z flag behaviour is useful for 32-bit arithmetic.

**Examples** `cmp.c.r %r1, %r2`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rs | | | | | Rt | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# cmp.i — Compare, immediate

**Instruction** `cmp.i Rs, #immediate`

**Description** A 16-bit compare of the register with an immediate

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == #immediate`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs - #immediate`

**Usage Notes** This is a 16-bit compare of a register with an immediate.

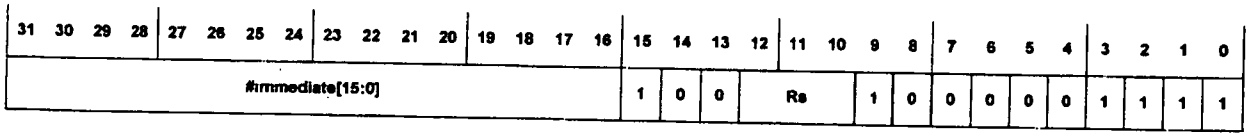
To compare a register against zero, use `cmp.r Rs, %r0`.

A number of commonly used combinations of registers and immediates are available in two 16-bit encodings of this instruction.

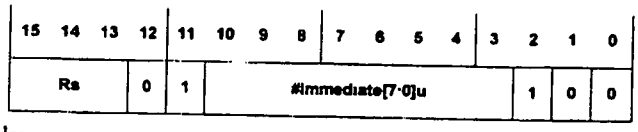
**Examples**

```
cmp.i %r1, #0x1234
cmp.i %r1, #0xFEDC
```

### 32-bit Encoding



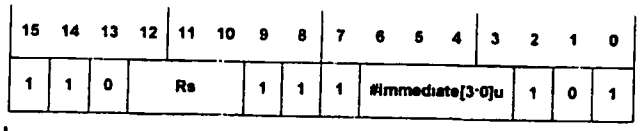

### 16-bit Encoding - 1




The following rules apply:

- In this encoding, Rs is in the range %r0 to %r6.
- This encoding can represent immediates in the range 0 to 255.

### 16-bit Encoding - 2




The following rules apply:

- The 3-bit Rs field can specify %r0 to %r6.

- The immediate value is (#immediate[3:0]u – 16) giving a range of -1, -2, ..., -15, -16.

Cambridge Consultants

# cmp.r — Compare, register

**Instruction** `cmp.r Rs, Rt`

**Description** Compares two registers

**Flags**

- **Z** Set if `Rs == Rt`; cleared otherwise
- **N** Set if the result of the subtraction is negative; cleared if the result is positive
- **C** Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
- **V** Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

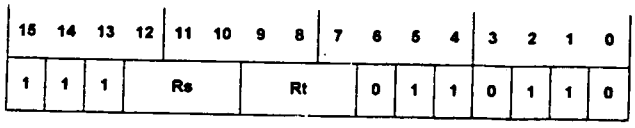
**Operation** `Rs - Rt`

**Usage Notes** This is a 16-bit compare of two registers.

**Examples**

```
cmp.r %r1, %r2
```

**16-bit Encoding**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rs | | | | Rt | | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

# cmp.x.i — Compare, exchange, immediate

**Instruction** `cmp.x.i Rs, #immediate`

**Description** A 16-bit compare of the register with an immediate and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if `Rs == #immediate`; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation** `Rs - #immediate`

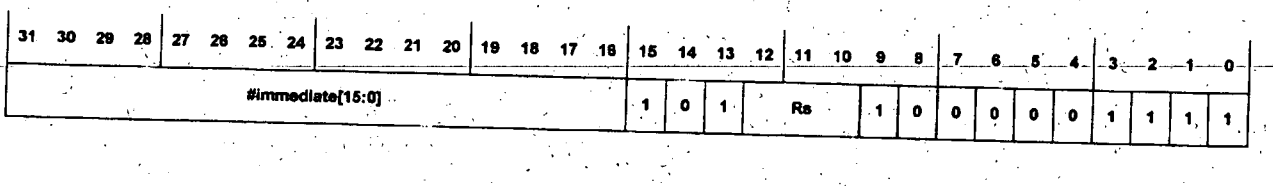
**Usage Notes** This is a 16-bit compare of an immediate with a register.

This is the same as `cmp.i` but with the operand order reversed.

**Examples**

```
cmp.x.i %r1, #0x1234
cmp.x.i %r1, #0xFEDC
```

**32-bit Encoding**



# cmp.xc.i Compare, with carry, exchange, immediate

**Instruction**

```
cmp.xc.i Rs, #immediate
```

**Description**

A 16-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result of the subtraction is negative; cleared if the result is positive |
| **C** | Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise |
| **V** | Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise |

**Operation**

```
#immediate - Rs - FLAGS.C
```

**Usage Notes**

This is a 16-bit compare of an immediate with a register and the carry bit.

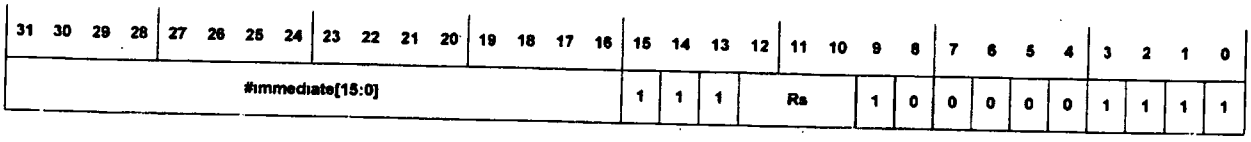
This is the same as `cmp.c.i` but with the operand order reversed.

The Z flag behaviour is useful for 32-bit arithmetic.

**Examples**

```
cmp.xc.i %r1, #0x1234
cmp.xc.i %r1, #0xFEDC
```

**32-bit Encoding**



# div.32s.i — Divide, 32-bit signed, immediate

**Instruction** `div.32s.i Rd, Rs, #immediate`

**Description** Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = {R(s+1), Rs} / #immediate`

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

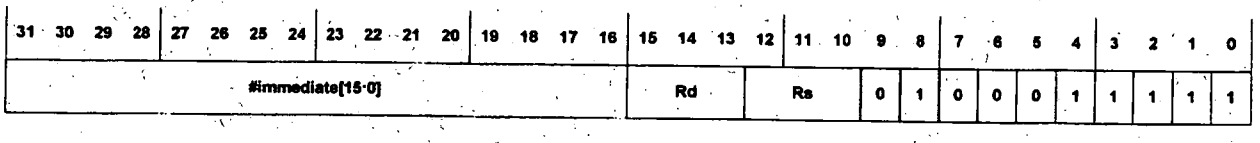
If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient is forced to zero.

Attempts to divide by zero operate as follows:

- The quotient is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples** `div.32s.i %r4, %r1, #20`

**32-bit Encoding**



# div.32s.r — Divide, 32-bit signed, register

**Instruction** `div.32s.r Rd, Rs, Rt`

**Description** Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = {R(s+1), Rs} / Rt`

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

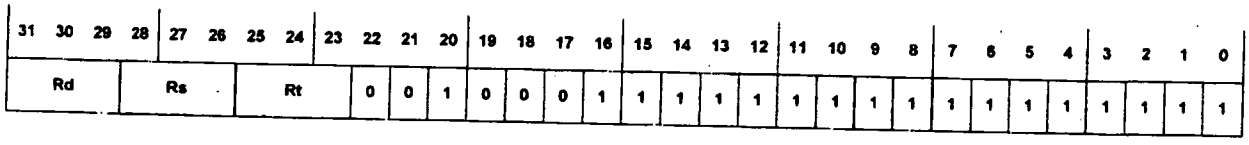
If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient is forced to zero.

Attempts to divide by zero operate as follows:

- The quotient is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples** `div.32s.r %r4, %r1, %r3`

**32-bit Encoding**



# div.32u.i — Divide, 32-bit unsigned, immediate

**Instruction**

```
div.32u.i Rd, Rs, #immediate
```

**Description**

Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of the result is 1; cleared otherwise |
| **C** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation**

```
Rd = {R(s+1), Rs} / #immediate
```

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient is forced to zero.

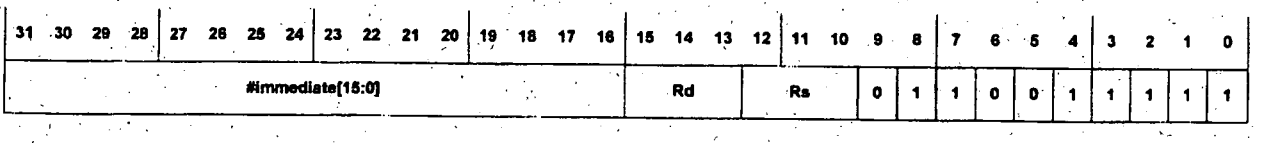
Attempts to divide by zero operate as follows:

- The quotient is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
div.32u.i %r4, %r1, #10
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | Rd | Rs | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

# div.32u.r — Divide, 32-bit unsigned, register

**Instruction**

```
div.32u.r Rd, Rs, Rt
```

**Description**

Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of the result is 1; cleared otherwise |
| **C** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation**

```
Rd = {R(s+1), Rs} / Rt
```

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient is forced to zero.

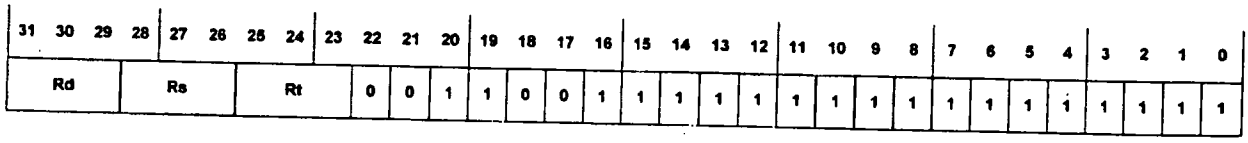
Attempts to divide by zero operate as follows:

- The quotient is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
div.32u.r %r4, %r1, %r3
```

**32-bit Encoding**



# div.s.i — Divide, signed, immediate

**Instruction** `div.s.i Rd, Rs, #immediate`

**Description** Signed 16-bit /16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | Set if the result is negative; cleared otherwise |
| C | Unchanged |
| V | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = Rs / #immediate`

**Usage Notes** Attempts to divide by zero operate as follows:

- The quotient is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The division -32768 / -1 gives a result of 0 and sets the overflow flag.

**Examples** `div.s.i %r1, %r2, #10`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15.0] | | | | | | | | | | | | | | | | Rd | | | | Rs | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

# div.s.r — Divide, signed, register

**Instruction** `div.s.r Rd, Rs, Rt`

**Description** Signed 16-bit / 16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = Rs / Rt`

**Usage Notes** Attempts to divide by zero operate as follows:

- The quotient is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The division -32768 / -1 gives a result of 0 and sets the overflow flag.

**Examples** `div.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# div.u.i — Divide, unsigned, immediate

**Instruction** `div.u.i Rd, Rs, #immediate`

**Description** Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient

**Flags**

- **Z** — Set if the result is zero; cleared otherwise
- **N** — Set if bit 15 of the result is 1; cleared otherwise
- **C** — Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise
- **V** — Unchanged

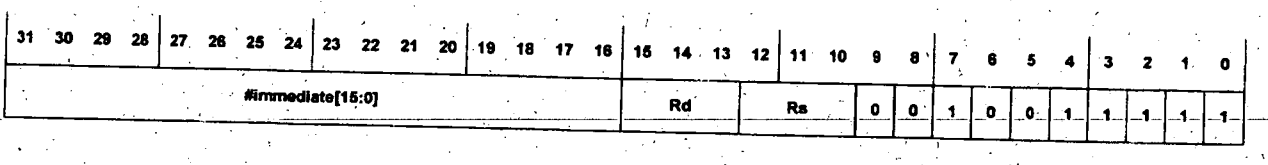
**Operation** `Rd = Rs / #immediate`

**Usage Notes** Attempts to divide by zero operate as follows:

- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples** `div.u.i %r1, %r2, #0x10`

**32-bit Encoding**



# div.u.r — Divide, unsigned, register

**Instruction** `div.u.r Rd, Rs, Rt`

**Description** Unsigned 16-bit /16-bit integer divide to give a 16-bit quotient

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 15 of the result is 1; cleared otherwise |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation** `Rd = Rs / Rt`

**Usage Notes** Attempts to divide by zero operate as follows:

- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
div.u.r %r1, %r2, %r3
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# divrem.32s.i Divide&remainder, 32-bit signed, immediate

**Instruction** `divrem32.s.i Rd, Rs, #immediate`

**Description** Signed 32-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation**

```
Rd     = {R(s+1), Rs} / #immediate
R(d+1) = {R(s+1), Rs} % #immediate
```

**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient and remainder are forced to zero.

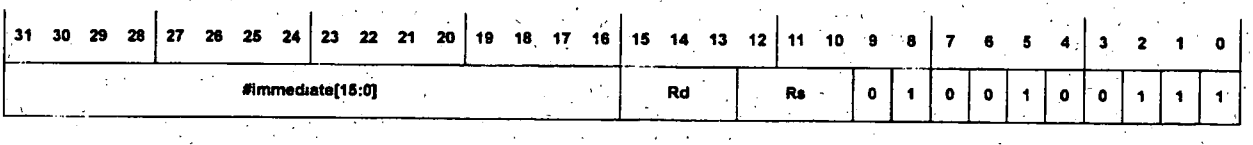
Attempts to divide by zero operate as follows:

- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
divrem32.s.i %r4, %r1, #10
```

**32-bit Encoding**



Cambridge Consultants

# divrem.32s.r Divide&remainder, 32-bit signed, register

**Instruction**

```
divrem.32s.r Rd, Rs, Rt
```

**Description**

Signed 32-bit /16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation**

```
Rd     = {R(s+1), Rs} / Rt
R(d+1) = {R(s+1), Rs} % Rt
```

**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient and remainder are forced to zero.

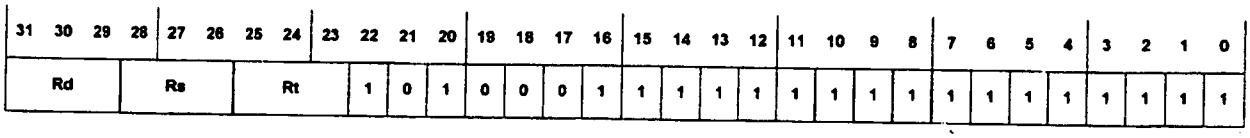
Attempts to divide by zero operate as follows:

- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
divrem.32s.r %r4, %r1, %r3
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| Rd | Rs | Rt | 1 0 1 | 0 0 0 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 |

# divrem.32u.i Divide&remainder, 32-bit unsigned, immediate

**Instruction** `divrem.32u.i Rd, Rs, #immediate`

**Description** Unsigned 32-bit /16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation**

```
Rd     = {R(s+1), Rs} / #immediate
R(d+1) = {R(s+1), Rs} % #immediate
```

**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient and remainder are forced to zero.

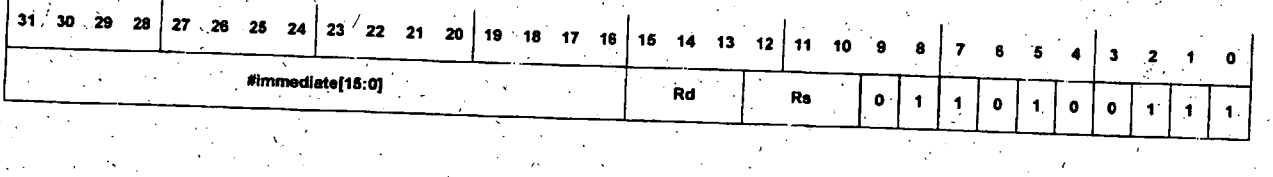
Attempts to divide by zero operate as follows:

- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
divrem.32u.i %r4, %r1, #immediate
```

**32-bit Encoding**



# divrem.32u.r Divide&remainder, 32-bit unsigned, register

**Instruction**

```
divrem.32u.r Rd, Rs, Rt
```

**Description**

Unsigned 32-bit /16-bit integer divide to give a 16-bit quotient and a 16-bit remainder.

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation**

```
Rd     = {R(s+1), Rs} / Rt
R(d+1) = {R(s+1), Rs} % Rt
```

**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient and remainder are forced to zero.

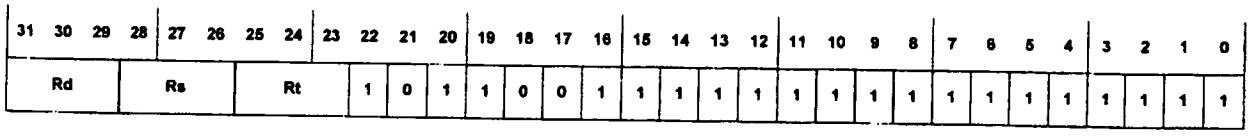
Attempts to divide by zero operate as follows:

- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
divrem.32u.r %r4, %r1, %r3
```

**32-bit Encoding**



# divrem.s.i — Divide&remainder, signed, immediate

**Instruction**

```
divrem.s.i Rd, Rs, #immediate
```

**Description**

Unsigned 16-bit / 16-bit integer divide to give a 16-bit integer quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation**

```
Rd     = Rs / #immediate
R(d+1) = Rs % #immediate
```

**Usage Notes**

Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

Attempts to divide by zero operate as follows:

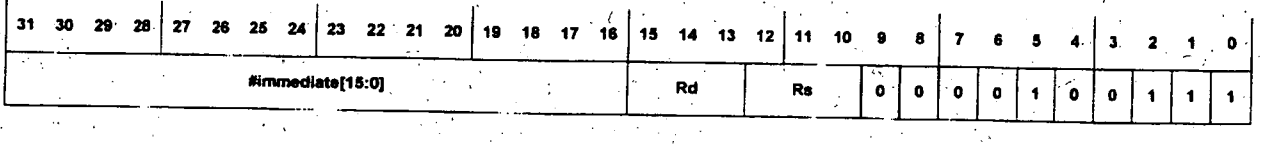
- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The division -32768 / -1 gives results of 0 and sets the overflow flag.

**Examples**

```
divrem.s.i %r1, %r2, #10
```

**32-bit Encoding**



# divrem.s.r — Divide&remainder, signed, register

**Instruction**

```
divrem.s.r Rd, Rs, Rt
```

**Description**

Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation**

```
Rd     = Rs / Rt
R(d+1) = Rs % Rt
```

**Usage Notes**

Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

Attempts to divide by zero operate as follows:

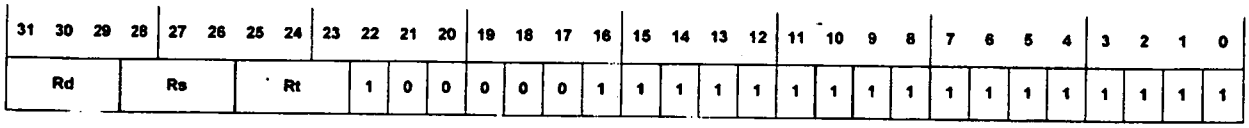
- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The division -32768 / -1 gives results of 0 and sets the overflow flag.

**Examples**

```
divrem.s.r %r1, %r2, %r3
```

**32-bit Encoding**



# divrem.u.i Divide&remainder, unsigned, immediate

**Instruction** `divrem.u.i Rd, Rs, #immediate`

**Description** Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| V | Unchanged |

**Operation**

```
Rd     = Rs / #immediate
R(d+1) = Rs % #immediate
```

**Usage Notes**

Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

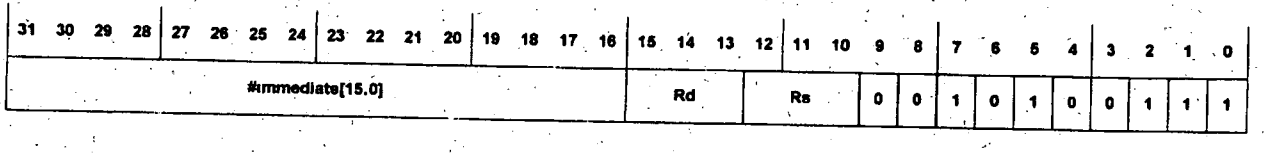
Attempts to divide by zero operate as follows:

- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
divrem.u.i %r1, %r2, #10
```

**32-bit Encoding**



# divrem.u.r Divide&remainder, unsigned, register

**Instruction** `divrem.u.r Rd, Rs, Rt`

**Description** Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation**

```
Rd     = Rs / Rt
R(d+1) = Rs % Rt
```

**Usage Notes**

Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

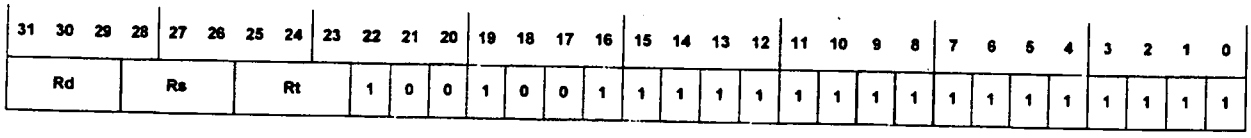
Attempts to divide by zero operate as follows:

- The quotient and remainder are zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
divrem.u.r %r1, %r2, %r3
```

**32-bit Encoding**



# halt

## Halt

**Instruction** `halt`

**Description** Stop the processor

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
PC = next instruction
if (FLAGS.Mode == UserMode) then
   halt processor
else
   throw IllegalInstruction exception
endif
```

**Usage Notes** An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

The processor is restarted using the SIF interface or a reset.

**Examples**

```
halt
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# irqd — Disable interrupts

**Instruction** `irqd`

**Description** Disable all maskable interrupts

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `FLAGS.E = 0`

**Usage Notes** An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

**Examples** `irqd`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# irqe — Enable interrupts

| | | |
|---|---|---|
| **Instruction** | | `irqe` |
| **Description** | | Enable maskable interrupts |
| **Flags** | **Z** | Unchanged |
| | **N** | Unchanged |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `FLAGS.E = 1` |
| **Usage Notes** | | An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode. |
| **Examples** | | `irqe` |

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# irqs — Interrupt status

| | | |
|---|---|---|
| **Instruction** | | `irqs Rd` |
| **Description** | | Read the interrupt enable bit |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Cleared to zero |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = FLAGS.E ? 1 : 0` |
| **Usage Notes** | | This instruction reads the interrupt enable bit in the `FLAGS` register. |
| **Examples** | | `irqs %r1` |

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rd | | | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

# ld.8z.i — Load, 8-bit, zero-extend, displacement

**Instruction** `ld.8z.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra), zero-extend from 8 to 16 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Cleared to zero |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = (uint16) (* (int8*) ( offset + Ra ) )`

**Usage Notes**

Ra and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

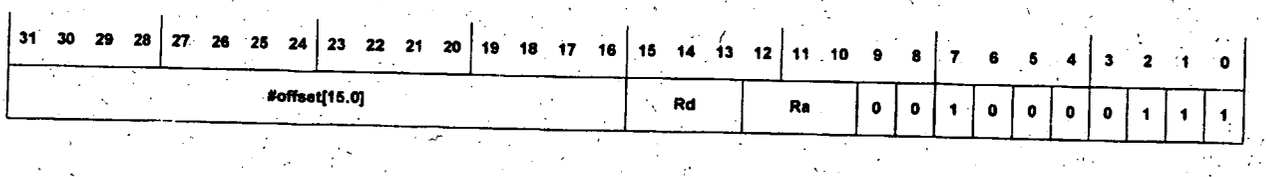
The memory value is zero-extended to 16 bits when written to Rd.

An 8-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

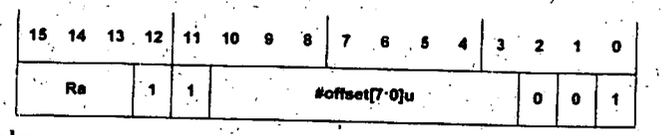
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `ld.8z.i %r1, @(2, %r7)`

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- Rd is %r0.
- This encoding can represent offsets in the range 0 to 255..

# ld.8z.p — Load, 8-bit, zero-extend, PC-relative

**Instruction** `ld.8z.p Rd, @(offset)`

**Description** Load Rd from address (offset + PC), zero-extend from 8 to 16 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Cleared to zero |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = (uint16) (* (int8*) ( offset + PC ) )
```

**Usage Notes**

PC and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

The memory value is zero-extended to 16 bits when written to Rd.

Rd must be in the range %r1 to %r7.

An 8-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
ld.8z.p %r1, @(357) // Stores R1 to (357 + PC)

ld.8z.p %r1, @(label) // Stores R1 to 'label'
```

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | Rd | 1 0 1 1 | 1 0 0 0 | 0 1 1 1 |

# ld.i — Load, displacement

**Instruction** `ld.i Rd, @(offset, Ra)`

**Description** Load Rd from address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = * (int16*) ( offset + Ra )`

**Usage Notes**

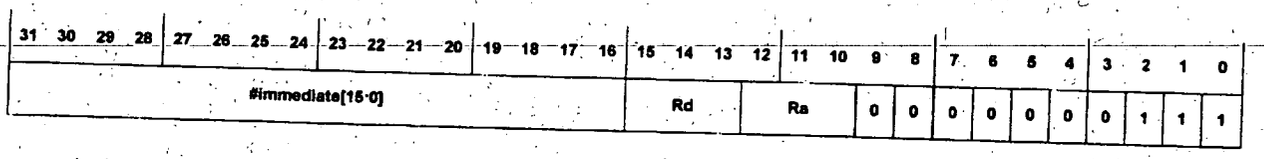
Ra and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

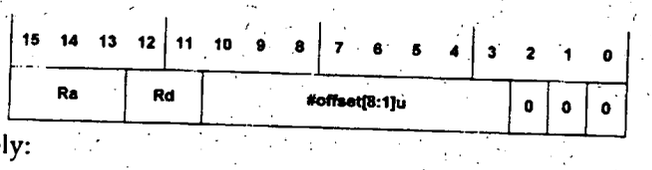
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `ld.i %r1, @(2, %r15)`

**32-bit Encoding**



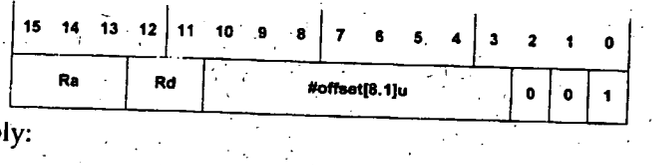

**16-bit Encoding - 1**




The following rules apply:

- The 2-bit Rd field can specify %r0, %r1, %r2 or %r3.
- This encoding can represent offsets in the range 0, 2, ..., 508, 510.

**16-bit Encoding - 2**




The following rules apply:

- The 2-bit Rd field can specify %r4, %r5, or %r6.

- This encoding can represent offsets in the range 0, 2, ..., 508, 510.

# ld.p — Load, PC-relative

**Instruction** `ld.p Rd, @(offset)`

**Description** Load Rd from address (offset + PC)

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | Set if the result is negative; cleared otherwise |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
Rd = * (int16*) ( offset + PC )
```

**Usage Notes**

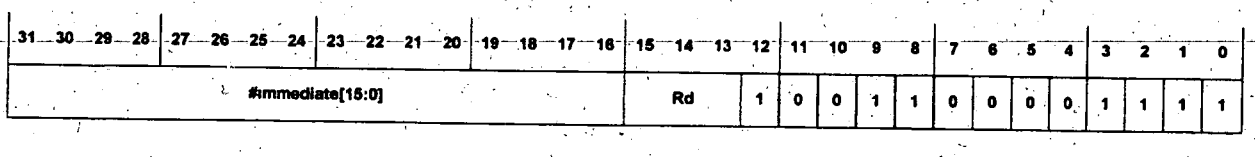
PC and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
ld.p %r1, @(357) // Stores R1 to (357 + PC)
ld.p %r2, @(label) // Stores R1 to 'label'
```

**32-bit Encoding**



# ld.r — Load, indexed

**Instruction** `ld.r Rd, @(Rx, Ra)`

**Description** Load Rd from address (2Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
Rd = * (int16*) ( 2*Rx + Ra )
```

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a 2-byte offset from Ra.

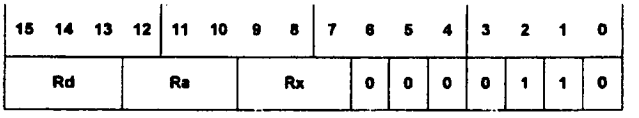
Rd must be in the range %r1 to %r7.

A 16-bit memory read is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
ld.r %r1, @(%r2, %r3)
```

**16-bit Encoding**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Ra | | | | Rx | | | | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

# lic — Read XAP4 licence number

**Instruction** `lic Rd`

**Description** Read the XAP4's licence number

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = XAP4 licence number`

**Usage Notes** The licence number is different for each customer chip design.

**Examples** `lic %r1`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rd | | | | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

# mov.32.r — move, accumulator

**Instruction** `mov.32.r Rd, Rs`

**Description** Accumulator-to-accumulator move

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

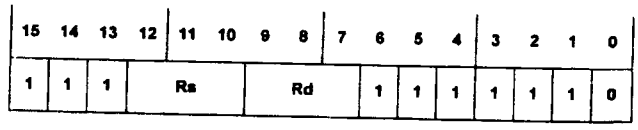
**Operation** `{R(d+1), Rd} = {R(s+1), Rs}`

**Usage Notes** Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

**Examples** `mov.32.r %r1, %r3`

**16-bit Encoding**



Cambridge Consultants

# mov.i — Move, immediate

**Instruction** `mov.i Rd, #immediate`

**Description** Move immediate to register

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | Set if the result is negative; cleared otherwise |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = #immediate`

**Usage Notes** For immediates in the range 0 to 255, and Rd in the range %r0 to %r6, a 16-bit encoding of this instruction is available.

**Examples**

```
mov.i %r1, #0x1234

mov.i %r1, #0xFEDC
```

## 32-bit Encoding

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | Rd | 0 0 1 1 | 0 0 0 0 | 1 1 1 1 |

## 16-bit Encoding - 1

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| Rd | 0 0 | #immediate[7:0]u | 1 0 0 |

The following rules apply:

- In this encoding, Rd is in the range %r0 to %r6.
- This encoding can represent immediate values in the range 0 to 255.

## 16-bit Encoding - 2

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|
| 1 1 0 1 | 1 1 Rd | #immediate[3:0]u | 1 0 0 |

The following rules apply:

- The 3-bit Rd field can specify %r0 to %r6.
- The immediate value is (#immediate[3:0]u – 16) giving a range of -1, -2, ..., -15, -16.

# mov.p — Move, PC-relative

**Instruction** `mov.p Rd, #label`

**Description** Calculate the absolute address of a label

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = #label`

**Usage Notes** This instruction calculates the absolute address of a label and is useful in systems where a program's text segment has been loaded to an address other than zero. The label would normally be a label in a text segment rather than a data segment.

The label can be between -32KByte to +32KByte relative to the current PC.

**Examples** This instruction calculates the absolute address of `label36`:

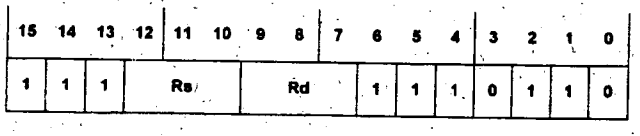
```
mov.p %r3, #label36
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #offset[15:0] | | | | | | | | | | | | | | | | Rd | | | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

# mov.r — move, register

| | | |
|---|---|---|
| **Instruction** | | `mov.r Rd, Rs` |
| **Description** | | Register-to-register move |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs` |
| **Usage Notes** | | – |
| **Examples** | | `mov.r %r1, %r2` |
| **16-bit Encoding** | | |



# movb2r — Move breakpoint to register

**Instruction** `movb2r Rd, BRs`

**Description** Transfer breakpoint register Bs to register Rd

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = BRs`

**Usage Notes** This instruction throws an `IllegalInstruction` exception if executed in User Mode.

To access the BRKE register, use the `movr2s` instruction.

**Examples** `movb2r %r1, %brk0`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | BRs | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# movr2b — Move register to breakpoint

**Instruction** `movr2b BRd, Rs`

**Description** Transfer a register to a breakpoint register

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

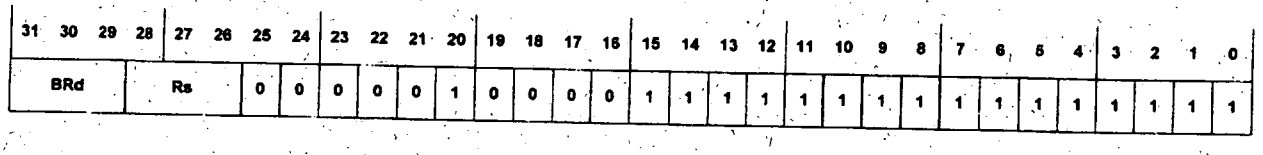
**Operation** `BRd = Rs`

**Usage Notes** This instruction throws an `IllegalInstruction` exception if executed in User Mode.

To access the BRKE register, use the `movr2s` instruction.

**Examples** `movr2b %brk0, %r1`

**32-bit Encoding**



# movr2s — Move register to special

**Instruction** `movr2s SRd, Rs`

**Description** Write to a special register

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `SRd = Rs`

**Usage Notes**

The `movs2r` and `movr2s` instructions allow Interrupt, Supervisor and NMI Mode access to the Special Registers.

The `movr2s` instruction can be used to change from any privileged mode to any mode.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples**

```
movr2s %flags_i, %r1
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SRd | | | | Rs | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# movs2r — Move special to register

**Instruction** `movs2r Rd, SRs`

**Description** Read from a special register

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero, cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = SRs`

**Usage Notes** The `movs2r` and `movr2s` instructions allow Interrupt, Supervisor and NMI Mode access to the Special Registers.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

**Examples** `movs2r %r1, %flags_i`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | SRs | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# mult.32s.i — Multiply, 32-bit signed, immediate

**Instruction** `mult.32s.i Rd, Rs, #immediate`

**Description** 16-bit by 16-bit signed integer multiply to give a 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} = Rs * #immediate`

**Usage Notes**

This is a signed 16x16 multiply, giving a 32-bit product in Rd.

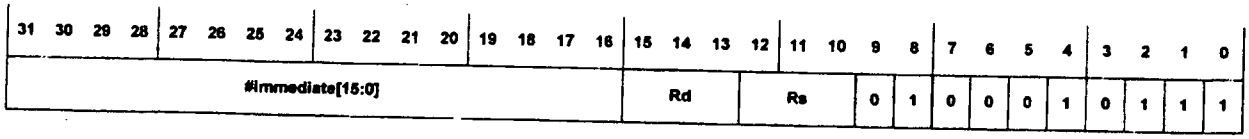
Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767.

**Examples** `mult.32s.i %r1, %r2, #0x1234`

**32-bit Encoding**



# mult.32s.r — Multiply, 32-bit signed, register

**Instruction** `mult.32s.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit signed integer multiply to give a 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} = Rs * Rt`

**Usage Notes**

This is a signed 16x16 multiply, giving a 32-bit product in Rd.

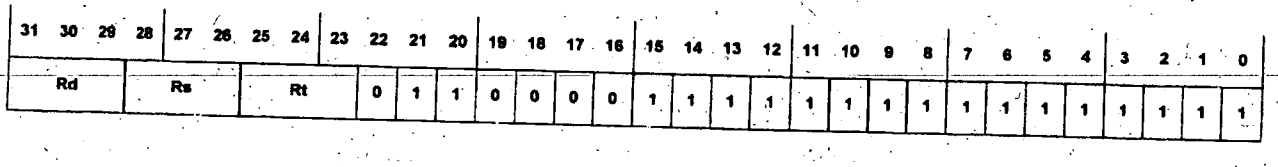
Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767.

**Examples** `mult.32s.r %r1, %r2, %r3`

**32-bit Encoding**



# mult.32u.i — Multiply, 32-bit unsigned, immediate

**Instruction** `mult.32u.i Rd, Rs, #immediate`

**Description** 16-bit by 16-bit signed integer multiply to give a 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of Rd is 1; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} = Rs * #immediate`

**Usage Notes**

This is an unsigned 16x16 multiply, giving a 32-bit product in Rd.

Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535.

**Examples** `mult.32u.i %r1, %r2, #0x1234`

**32-bit Encoding**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #immediate[15.0] | | | | Rd | Rs | 0 1 | 1 0 0 1 | 0 1 1 1 |

# mult.32u.r — Multiply, 32-bit unsigned, register

**Instruction** `mult.32u.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit unsigned integer multiply to give a 32-bit result

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of Rd is 1; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `{R(d+1), Rd} = Rs * Rt`

**Usage Notes**

This is an unsigned 16x16 multiply, giving a 32-bit product in Rd.

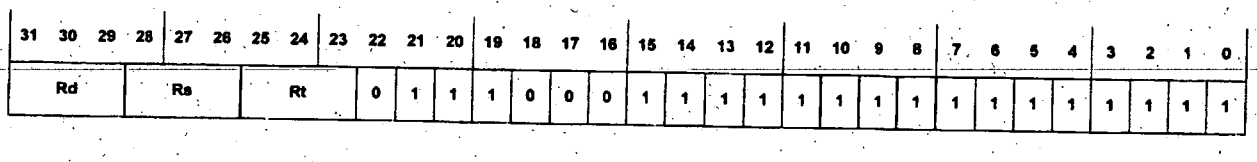
Rd must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd throw an `UnknownInstruction` exception.

Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535.

**Examples** `mult.32u.r %r1, %r2, %r3`

**32-bit Encoding**



# mult.i — Multiply immediate

| | | |
|---|---|---|
| **Instruction** | | `mult.i Rd, Rs, #immediate` |
| **Description** | | 16-bit by 16-bit integer multiply to give a 16-bit result |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs * #immediate` |
| **Usage Notes** | | This is a 16x16 multiply, giving the low 16 bits of the product in Rd. |
| **Examples** | | `mult.i %r1, %r2, #0x1234` |
| | | `mult.i %r1, %r2, #0xFEDC` |

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15.0] | | | | | | | | | | | | | | | | Rd | | | | Rs | | | | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

# mult.r — Multiply, register

| | | |
|---|---|---|
| **Instruction** | | `mult.r Rd, Rs, Rt` |
| **Description** | | 16-bit by 16-bit integer multiply to give low 16-bits of the result |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs * Rt` |
| **Usage Notes** | | This is a 16x16 multiply, giving the low 16 bits of the product in Rd.<br>Rd, Rs and Rt must be in the range %r0 to %r6. |
| **Examples** | | `mult.r %r1, %r2, %r3` |

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# mult.sh.r — Multiply, signed shift right, register

**Instruction**

```
mult.sh.r Rd, Rs, Rt, #immediate
```

**Description**

16-bit by 16-bit signed integer multiply to give a 32-bit result which is then signed-shifted right by the immediate

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if the result is incorrect; cleared otherwise |

**Operation**

```
Rd = (Rs * Rt) >> #immediate
```

**Usage Notes**

The 16x16 signed multiply of Rs and Rt produces a 32-bit product in a temporary register. The 32-bit product is signed-shifted right by the immediate operand. This shift, like `shiftr.32s.i`, preserves the sign of the product by re-inserting the sign bit into the vacated most-significant bit.

Finally, the low 16 bits are written to Rd.

This instruction is useful when operating on fixed-point numbers, where a normalising shift is needed after a multiplication.

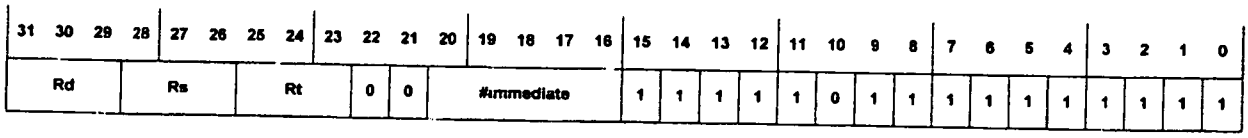
The V flag is set if the result is incorrect:

$$V = ( Rd * 2^{\#immediate} != (Rs * Rt) )$$

**Examples**

```
mov.i  %r1, #0x3733      // +3.45 in signed 4.12 format
mov.i  %r2, #0xEC52      // -1.23 in signed 4.12 format
mult.sh.r %r3, %r1, %r2, #(16-4)
// Here, %r3 is 0xBC1B, or -4.24 in 4.12 format
```

**32-bit Encoding**



# nop — No operation

| | | |
|---|---|---|
| **Instruction** | | nop |
| **Description** | | No operation |
| **Flags** | **Z** | Unchanged |
| | **N** | Unchanged |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | - |
| **Usage Notes** | | - |
| **Examples** | | nop |

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# or.i — Or, immediate

| | | |
|---|---|---|
| **Instruction** | | `or.i Rd, Rs, #immediate` |
| **Description** | | Logical OR of Rs with an immediate value |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs | #immediate` |
| **Usage Notes** | | – |
| **Examples** | | `or.i %r1, %r2, #0x1234` |
| | | `or.i %r1, %r2, #0xFEDC` |

### 32-bit Encoding

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15.0] | | | | | | | | | | | | | | | | Rd | | | | Rs | | | | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

### 16-bit Encoding

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 1 | Rd | | #immediate[3:0]u | | | | 1 | 0 | 1 | |

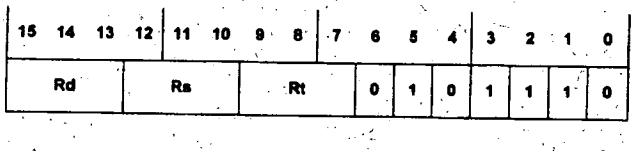
The following rules apply:

- The 3-bit Rd field can specify %r0 to %r6.
- This encoding can represent immediates in the range 0, 1, ..., 14, 15.

## or.r — Or, register

| | | |
|---|---|---|
| **Instruction** | | `or.r Rd, Rs, Rt` |
| **Description** | | Logical OR of two registers |
| **Flags** | **Z** | Set if the result is zero; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs \| Rt` |
| **Usage Notes** | | Rd, Rs and Rt must be in the range %r0 to %r6. |
| **Examples** | | `or.r %r1, %r2, %r3` |

**16-bit Encoding**



# pop — Pop from stack

**Instruction** `pop RegList, #offset`

**Description** Pop registers from stack and load a return value in R0

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** -

**Usage Notes**

This instruction can be used to close a stack frame. In addition to popping registers off the stack, the stack pointer can be increased by a further amount to remove the callee's local variables.

The RegList operand specifies which registers are popped and can contain registers in the range R0 to R6. Refer to section 5.10, "Register Lists" for details of RegList specifications.

The offset can take values in the range 0, +2, ..., +30.

The operation sequence is:

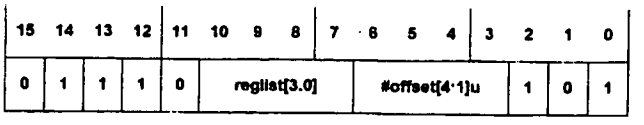
- Registers R0 to R6 are loaded from the stack, as specified in RegList. Lower numbered registers are loaded first and from lower stack addresses. Higher numbered registers are loaded last and from higher memory addresses.
- The stack pointer is increased by `#offset`+2n, where n is the number of selected registers in RegList.

A `NullPointer` exception is thrown if the stack pointer ascends to zero or to within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU. An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned.

**Examples**

```
pop   {%r3-%r6}, #0
pop   {%r3-%r6}, #4
```

**16-bit Encoding**



# pop.ret — Pop from stack and return

**Instruction** `pop.ret RegList, #offset`

**Description** Pop registers from stack and return with a return value in R0

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

**Usage Notes**

This instruction is identical to pop but additionally returns from a subroutine.

In addition to popping registers off the stack, the stack pointer can be increased by a further amount to remove the callee's local variables.

The RegList operand specifies which registers are popped and can contain registers in the range R0 to R6. Refer to section 5.10, "Register Lists" for details of RegList specifications.

The offset can take values in the range 0, +2, ..., +30.

The operation sequence is:

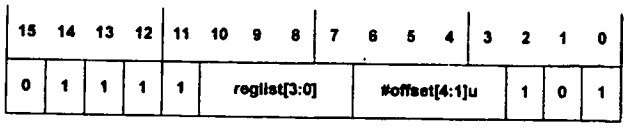
- Registers R0 to R6 are loaded from the stack, as specified in RegList. Lower numbered registers are loaded first and from lower stack addresses. Higher numbered registers are loaded last and from higher memory addresses
- The stack pointer is increased by `#offset`+2n, where n is the number of selected registers in RegList.
- Register R6 (the Link Register) is copied to the Program Counter. This is done even if R6 is not in RegList. If %r6 is in RegList, the popped value is copied to the Program Counter.

A `NullPointer` exception is thrown if the stack pointer ascends to zero or to within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU. An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned. An `IllegalCodeAddress` exception is thrown if bit 0 of R14 is not zero.

**Examples**

```
pop.ret {%r3-%r6}, #0
pop.ret {%r3-%r6}, #4
```

**16-bit Encoding**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | reglist[3:0] | | | #offset[4:1]u | | | | 1 | 0 | 1 | |

Cambridge Consultants

# print.r — Print, register

**Instruction** `print.r Rs`

**Description** Print a register (used in conjunction with a debugger)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** –

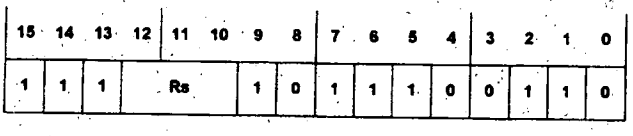
**Usage Notes** The xIDE simulator prints value the in Rs[7:0], interpreted as a character. This can be used for generating `putchar()` output in the debugger.

Hardware implementations of XAP4 treat this instruction as a nop.

**Examples**

```
print.r %r1
```

**16-bit Encoding**



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rs | | | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

# push — Push to stack

**Instruction** `push RegList, #offset`

**Description** Push registers onto the stack

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** -

**Usage Notes**

In addition to pushing registers to the stack, the stack pointer can be decreased by a further amount to create a stack frame for the callee's local variables.

The RegList operand specifies which registers are to be loaded and can contain registers in the range R0 to R6. Refer to section 5.10, "Register Lists" for details of RegList specifications.

The offset can take values in the range 0, +2, ..., +30.

The operation sequence is:

- Decrease the Stack Pointer (R7) by `#offset`+2n, where n is the number of selected registers.
- Store the selected registers to memory. Higher numbered registers are stored first and to higher memory addresses. Lower numbered registers are stored last and to lower stack addresses.

A `NullPointer` exception is thrown if the stack pointer descends to zero or within the MMU's defintion of a null pointer. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU. An `IllegalStackAddress` exception is thrown if the stack pointer is not word-aligned.

**Examples**

```
push {%r3-%r6}, #4
push {%r3-%r6}, #0
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | reglist[3:0] | | | #offset[4:1]u | | | | 1 | 0 | 1 | |

# rem.32s.i Remainder, 32-bit signed, immediate

**Instruction** `rem.32s.i Rd, Rs, #immediate`

**Description** 32-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = {R(s+1), Rs} % #immediate`

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

If the (discarded) quotient cannot be represented in 16 bits, the overflow flag is set.

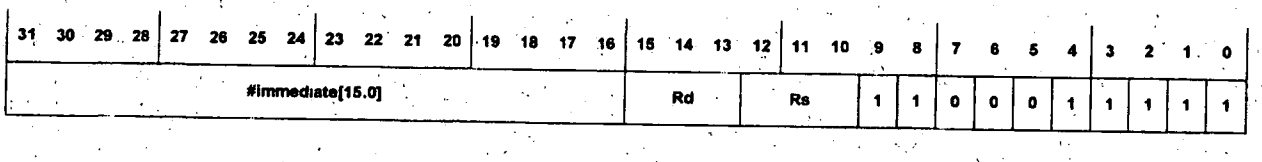
Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The remainder -32768 % -1 gives a result of 0 and sets the overflow flag.

**Examples** `rem.32s.i %r4, %r1, #10`

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #immediate[15.0] | | | | Rd | Rs 1 1 | 0 0 0 1 | 1 1 1 1 |

# rem.32s.r — Remainder, 32-bit signed, register

**Instruction** `rem.32s.r Rd, Rs, Rt`

**Description** 32-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = {R(s+1), Rs} % Rt`

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

If the (discarded) quotient cannot be represented in 16 bits, the overflow flag is set.

Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The remainder -32768 % -1 gives a result of 0 and sets the overflow flag.

**Examples** `rem.32s.r %r4, %r1, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Cambridge Consultants

# rem.32u.i — Remainder, 32-bit unsigned, immediate

**Instruction** `rem.32u.i Rd, Rs, #immediate`

**Description** 32-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of the result is 1; cleared otherwise |
| **C** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation** `Rd = {R(s+1), Rs} % #immediate`

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

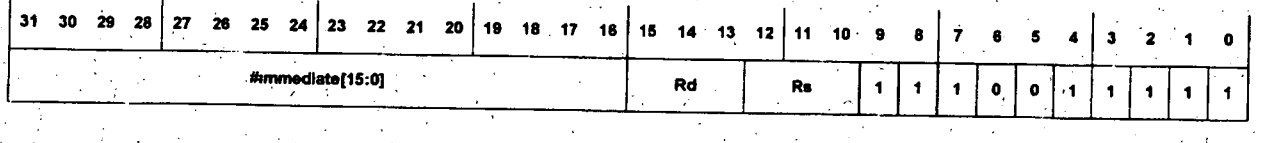
If the (discarded) quotient cannot be represented in 16 bits, the carry flag is set.

Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples** `rem.32u.i %r3, %r1, #10`

**32-bit Encoding**



# rem.32u.r Remainder, 32-bit unsigned, register

**Instruction** `rem.32u.r Rd, Rs, Rt`

**Description** 32-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 31 of the result is 1; cleared otherwise |
| **C** | Set if an overflow occurs or if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

**Operation** `Rd = {R(s+1), Rs} % Rt`

**Usage Notes**

Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rs throw an `UnknownInstruction` exception.

If the (discarded) quotient cannot be represented in 16 bits, the carry flag is set.

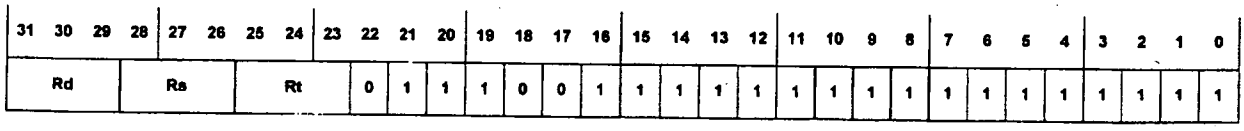
Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples**

```
rem.32u.r %r3, %r1, #10
```

**32-bit Encoding**



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# rem.s.i — Remainder, signed, immediate

**Instruction** `rem.s.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = Rs % #immediate`

**Usage Notes** Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The remainder -32768 % -1 gives a result of 0 and sets the overflow flag.

**Examples** `rem.s.i %r1, %r2, #10`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | | | | | | | | | | | | | Rd | | | | Rs | | | | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

Cambridge Consultants

# rem.s.r — Remainder, signed, register

**Instruction** `rem.s.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Set if an overflow occurs or an attempt is made to divide by zero in a privileged mode; cleared otherwise |

**Operation** `Rd = Rs % Rt`

**Usage Notes** Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

The remainder -32768 % -1 gives a result of 0 and sets the overflow flag.

**Examples** `rem.s.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Cambridge Consultants

# rem.u.i — Remainder, unsigned, immediate

**Instruction** `rem.u.i Rd, Rs, #immediate`

**Description** 16-bit by 16-bit signed divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 15 of the result is 1; cleared otherwise |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

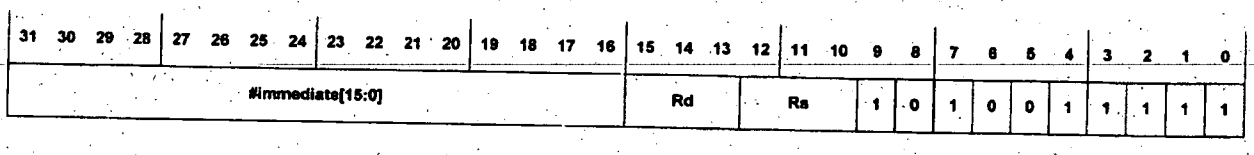
**Operation** `Rd = Rs % #immediate`

**Usage Notes** Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples** `rem.u.i %r1, %r2, #10`

**32-bit Encoding**



Cambridge Consultants

# rem.u.r — Remainder, unsigned, register

**Instruction** `rem.u.r Rd, Rs, Rt`

**Description** 16-bit by 16-bit unsigned divide to give a 16-bit remainder

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if bit 15 of the result is 1; cleared otherwise |
| **C** | Set if an attempt is made to divide by zero in a privileged mode; cleared otherwise |
| **V** | Unchanged |

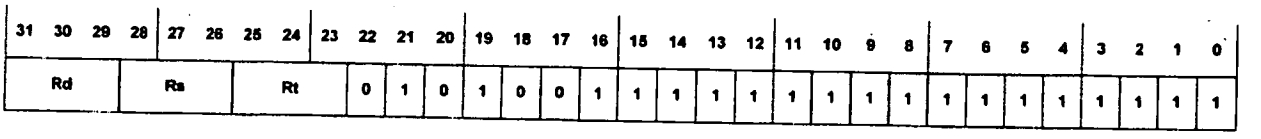
**Operation** `Rd = Rs % Rt`

**Usage Notes** Attempts to divide by zero operate as follows:

- The remainder is zero.
- In User Mode, the processor throws a `DivideByZero` exception.
- In a privileged mode, the division does not take place and the destination register remains unchanged.

**Examples** `rem.u.r %r1, %r2, %r3`

**32-bit Encoding**



Cambridge Consultants

# rotatel.32.i — Rotate left, 32-bit, immediate

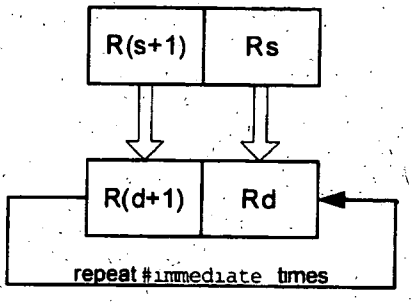
**Instruction** `rotatel.32.i Rd, Rs, #immediate`

**Description** 32-bit rotate left

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | The N bit is the final value of bit 31 of Rd |
| C | The C bit is the final value of bit 0 of Rd |
| V | Unchanged |

**Operation**




**Usage Notes**

The rotation is a 32-bit rotation and does not rotate through the carry bit.

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The immediate value must be between 0 and 31.

The flags are not modified if the rotate count is zero.

**Examples** `rotatel.32.i %r1, %r2, #10`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | 0 | 1 | 1 | 0 | 1 | #immediate[4.0] | | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Cambridge Consultants

# rotatel.32.r — Rotate left, 32-bit, register

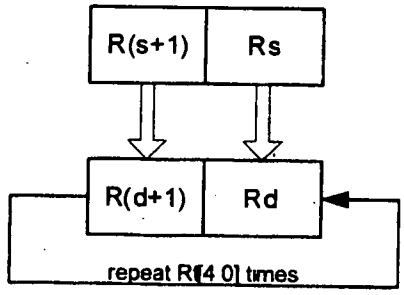
**Instruction** `rotatel.32.r Rd, Rs, Rt`

**Description** Rotate left

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit is the final value of bit 0 of Rd |
| **V** | Unchanged |

**Operation**




**Usage Notes**

The rotation is a 32-bit rotation and does not rotate through the carry bit.

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The flags are not modified if the rotate count is zero.

**Examples** `rotatel.32.r %r1, %r2, %r3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Cambridge Consultants

# rotatel.i — Rotate left, immediate

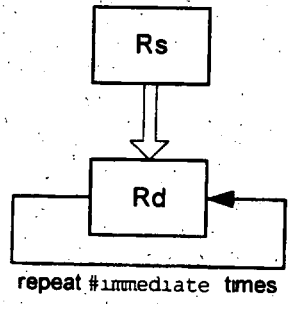
**Instruction** `rotatel.i Rd, Rs, #immediate`

**Description** Rotate left

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | The N bit is the final value of bit 15 of Rd |
| C | The C bit is the final value of bit 0 of Rd |
| V | Unchanged |

**Operation**




**Usage Notes**

The rotation is a 16-bit rotation and does not rotate through the carry bit.

The immediate value must be between 0 and 15.

Rd and Rs must be in the range %r0 to %r6.

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The flags are not modified if the rotate count is zero.

**Examples**

```
rotatel.i %r1, %r2, #10
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rs | | | | Rd | | | | #immediate[3:0] | | 1 | 0 | 1 |

# rotatel.r — Rotate left, register

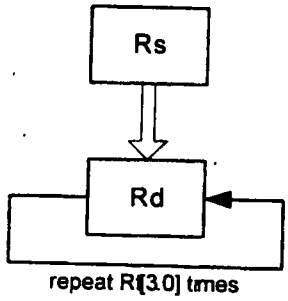
**Instruction** `rotatel.r Rd, Rs, Rt`

**Description** Rotate left

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 15 of Rd |
| **C** | The C bit is the final value of bit 0 of Rd |
| **V** | Unchanged |

**Operation**




**Usage Notes**

The rotation is a 16-bit rotation and does not rotate through the carry bit.

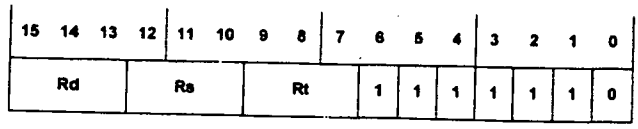
Rd, Rs and Rt must be in the range %r0 to %r6.

The flags are not modified if the rotate count is zero.

**Examples**

```
rotatel.r %r1, %r2, %r3
```

**16-bit Encoding**



# rtie — Return from interrupt/exception

**Instruction** `rtie`

**Description** Return from interrupt or exception

**Flags**

| | |
|---|---|
| **Z** | Restored from SFLAGS |
| **N** | Restored from SFLAGS |
| **C** | Restored from SFLAGS |
| **V** | Restored from SFLAGS |

**Operation**

```
if (FLAGS.Mode == NMIMode) then
   FLAGS = SFLAGS_N
   PC    = R6_N
elseif (FLAGS.Mode == InterruptMode) then
   FLAGS = SFLAGS_I
   PC    = R6_I
elseif (FLAGS.Mode == SupervisorMode) then
   FLAGS = SFLAGS_S
   PC    = R6_S
endif
```

**Usage Notes**

The null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code.

If nested interrupts can occur, it is necessary to save and restore SFLAGS_I within the ISR.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

This instruction throws an `IllegalCodeAddress` exception if bit 0 of the Link Register is not zero.

**Examples**

```
rtie
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# sext.r — Sign extend, register

**Instruction** `sext.r Rd`

**Description** Sign extend from 8 bits to 16 bits

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd[15:8] = Rd[7]`

**Usage Notes** This instruction sign-extends an 8-bit value by copying the sign bit into the top 8 bits.

**Examples** `sext %r1`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rd | | | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

# shiftl.32.i — Shift left, 32-bit, immediate

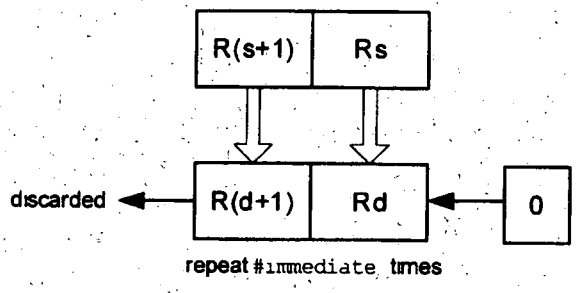
**Instruction** `shiftl.32.i Rd, Rs, #immediate`

**Description** 32-bit left shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

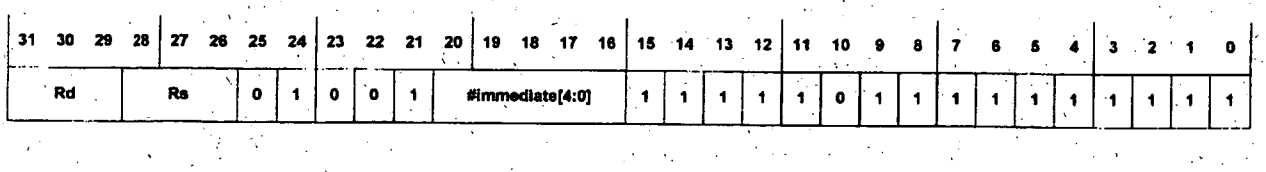
The immediate can take values in the range 0 to 31.

The vacated least-significant bits are filled with the zeros.

The flags are not modified if the shift count is zero.

**Examples** `shiftl.32.i %r1, %r3, #3`

**32-bit Encoding**



# shiftl.32.r — Shift left, 32-bit, register

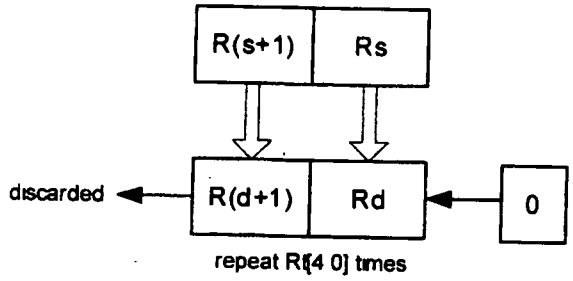
**Instruction** `shiftl.64.r Rd, Rs, Rt`

**Description** 32-bit left shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

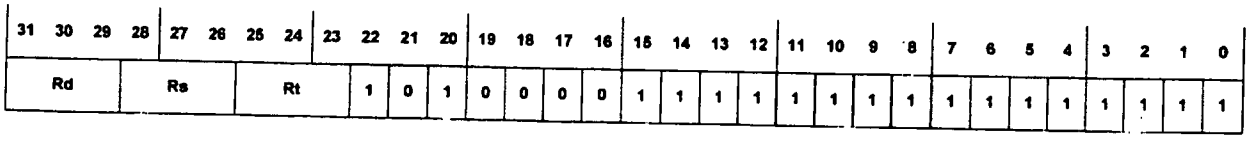
Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The vacated least-significant bits are filled with the zeros.

The flags are not modified if the shift count is zero.

**Examples** `shiftl.32.r %r1, %r3, %r5`

**32-bit Encoding**



Cambridge Consultants

# shiftl.i — Shift left, immediate

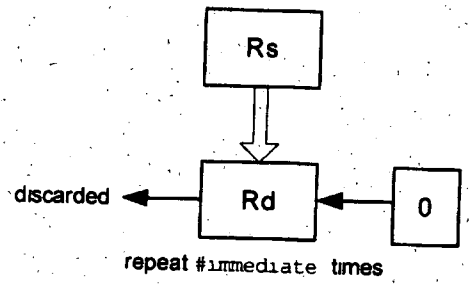
**Instruction** `shiftl.i Rd, Rs, #immediate`

**Description** Shift left

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | The N bit is the final value of bit 15 of Rd |
| C | The C bit contains the last bit shifted out |
| V | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must be in the range %r0 to %r6.

The immediate can take values in the range 0 to 15.

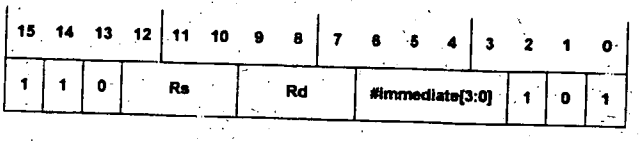
The vacated least-significant bits are filled with the zeros.

The flags are not modified if the shift count is zero.

**Examples**

```
shiftl.i %r1, %r2, #10
```

**16-bit Encoding**



# shiftl.r — Shift left, register

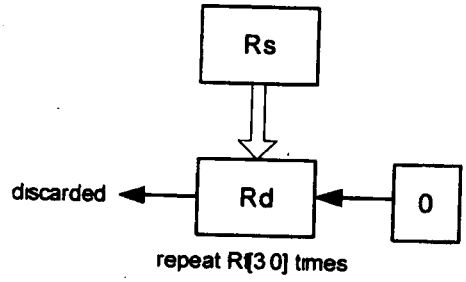
**Instruction** `shiftl.r Rd, Rs, Rt`

**Description** Shift left

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 15 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd, Rs and Rt must be in the range %r0 to %r6.

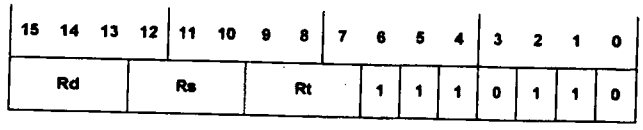
The vacated least-significant bits are filled with the zeros.

The flags are not modified if the shift count is zero.

**Examples**

```
shiftl.r %r1, %r2, %r3
```

**16-bit Encoding**



# shiftr.32s.i — Shift right, 32-bit signed, immediate

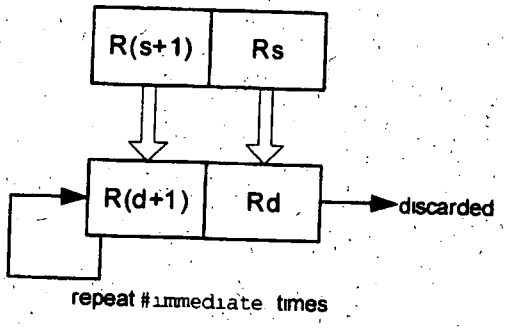
**Instruction** `shiftr.32s.i Rd, Rs, #immediate`

**Description** 32-bit signed right shift

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | The N bit is the final value of bit 31 of Rd |
| C | The C bit contains the last bit shifted out |
| V | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The immediate can take values between 0 and 31.

The vacated most-significant bits are filled with the sign bit.

The flags are not modified if the shift count is zero.

**Examples** `shiftr.32s.i %r1, %r3, #3`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | 0 | 0 | 0 | 0 | 1 | #immediate[4.0] | | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# shiftr.32s.r — Shift right, 32-bit signed, register

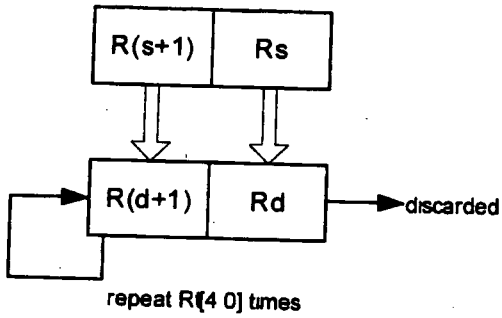
**Instruction** `shiftr.64s.r Rd, Rs, Rt`

**Description** 32-bit signed right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

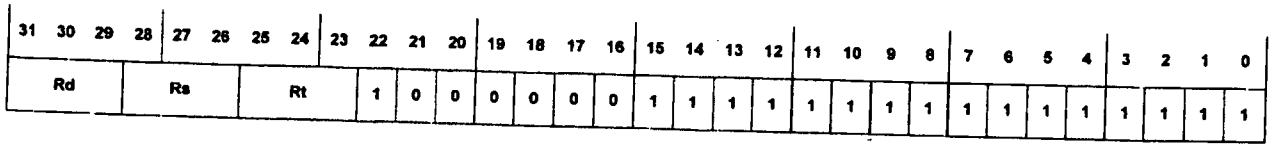
Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The vacated most-significant bits are filled with the sign bit.

The flags are not modified if the shift count is zero.

**Examples** `shiftr.32s.r %r1, $r3, %r5`

**32-bit Encoding**



# shiftr.32u.i — Shift right, 32-bit unsigned, immediate

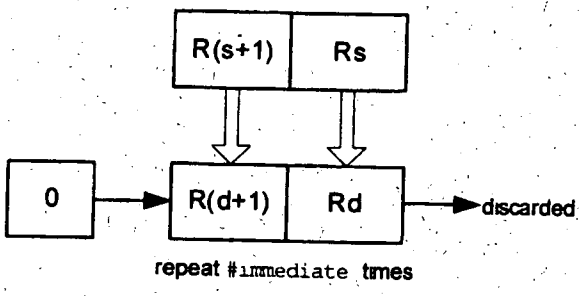
**Instruction** `shiftr.32u.i Rd, Rs, #immediate`

**Description** 32-bit signed right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The immediate can take values between 0 and 31.

The vacated most-significant bits are filled with zeros.

The flags are not modified if the shift count is zero.

**Examples**

```
shiftr.32u.i %r1, %r3, #3
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | 0 | 0 | 1 | 0 | 1 | #immediate[4.0] | | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## shiftr.32u.r — Shift right, 32-bit unsigned, register

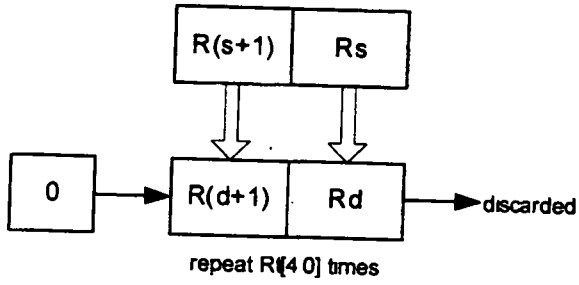
**Instruction** `shiftr.32u.r Rd, Rs, Rt`

**Description** 32-bit unsigned right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 31 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must specify one of the 32-bit accumulators (%r0 to %r5).

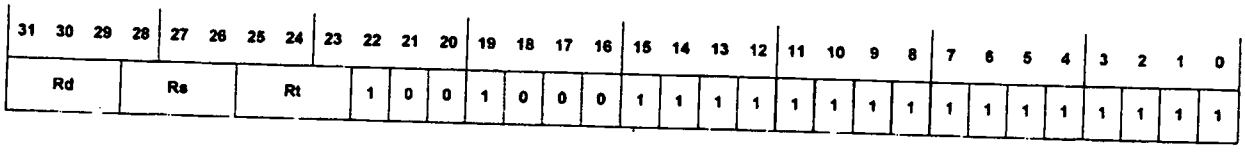
Other registers for Rd or Rs throw an `UnknownInstruction` exception.

The vacated most-significant bits are filled with zeros.

The flags are not modified if the shift count is zero.

**Examples** `shiftr.32u.r %r1, %r3, %r5`

**32-bit Encoding**



# shiftr.s.i — Shift right, signed, immediate

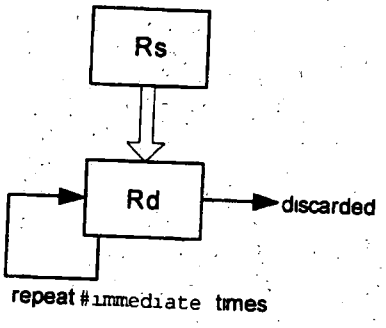
**Instruction** `shiftr.s.i Rd, Rs, #immediate`

**Description** 16-bit signed right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 15 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must be in the range %r0 to %r6.

The immediate can take values between 0 and 15.

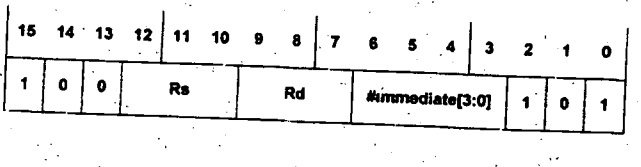
The vacated most-significant bits are filled with the sign bit.

The flags are not modified if the shift count is zero.

**Examples**

```
shiftr.s.i %r1, %r3, #3
```

**16-bit Encoding**



# shiftr.s.r — Shift right, signed, register

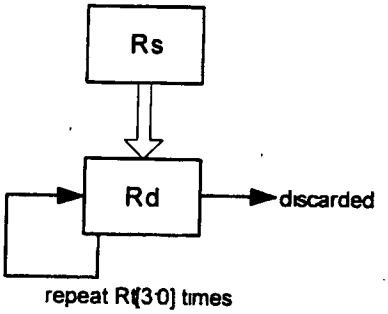
**Instruction** `shiftr. s.r Rd, Rs, Rt`

**Description** 16-bit signed right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 15 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd, Rs and Rt must be in the range %r0 to %r6.

The vacated most-significant bits are filled with the sign bit.

The flags are not modified if the shift count is zero.

**Examples** `shiftr.s.r %r1, %r3, %r5`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

## shiftr.u.i — Shift right, unsigned, immediate

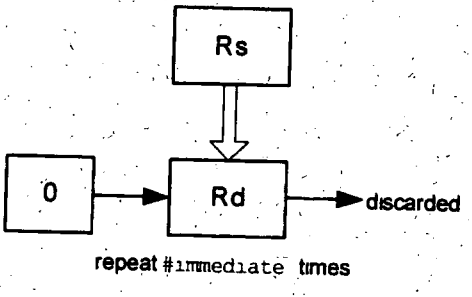
**Instruction** `shiftr.u.r Rd, Rs, #immediate`

**Description** 16-bit unsigned shift right

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 15 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd and Rs must be in the range %r0 to %r6.

The immediate value must be between 0 and 15.

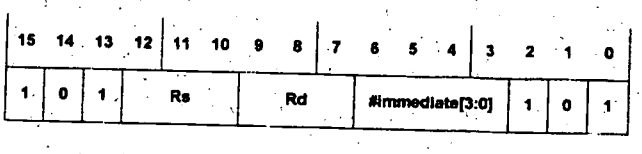
The vacated most-significant bits are filled with the zeros.

The flags are not modified if the shift count is zero.

**Examples**

```
shiftr.u.i %r1, %r2, #3
```

**16-bit Encoding**



## shiftr.u.r — Shift right, unsigned, register

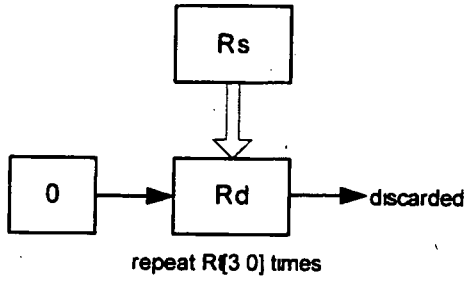
**Instruction** `shift.u.r Rd, Rs, Rt`

**Description** 16-bit unsigned right shift

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | The N bit is the final value of bit 15 of Rd |
| **C** | The C bit contains the last bit shifted out |
| **V** | Unchanged |

**Operation**




**Usage Notes**

Rd, Rs and Rt must be in the range %r0 to %r6.

The vacated most-significant bits are filled with zeroes.

The flags are not modified if the shift count is zero.

**Examples**

```
shiftr.u.r %r1, %r2, %r3
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

**Instruction** `sif`

**Description** Perform SIF cycle

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
if (FLAGS.Mode == UserMode) then
   allow SIF access
else
   throw IllegalInstruction exception
endif
```

**Usage Notes**

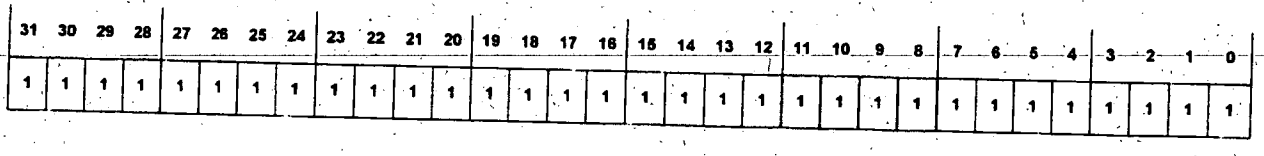
The `sif` instruction allows the XAP4 to perform a SIF cycle. Two SIF cycles are needed to access unaligned data objects in memory.

This instruction throws an `IllegalInstruction` exception if executed in User Mode.

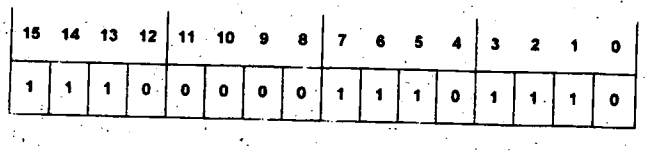
**Examples**

```
sif
```

**32-bit Encoding**




**16-bit Encoding**



# sleepnop

## Sleep

**Instruction** `sleepnop`

**Description** Put the XAP4 into NOP Sleep state

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
if (FLAGS.Mode == UserMode) then
   throw IllegalInstruction exception
else
   put XAP4 into NOP Sleep state
endif
```

**Usage Notes**

Put the XAP4 into the NOP Sleep state. SIF cycles are not allowed in the NOP Sleep state. xIDE is unable to gain access to XAP4 in this state.

An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

**Examples**

```
sleepnop
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# sleepsif — Sleep and allow SIF

**Instruction** `sleepsif`

**Description** Put the XAP4 into SIF Sleep state

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
if (FLAGS.Mode == UserMode) then
   throw IllegalInstruction exception
else
   put XAP4 into SIF Sleep mode
endif
```

**Usage Notes**

Put the XAP4 into the SIF Sleep state. SIF cycles are allowed in the SIF Sleep state.

An `IllegalInstruction` exception is thrown if this instruction is executed in User Mode.

**Examples** `sleepsif`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

# st.8.i — Store, 8-bit, displacement

**Instruction** `st.8.i Rs, @(offset, Ra)`

**Description** Store 8-bit value to address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `(* (int8*) ( offset + Ra ) ) = Rs[7:0]`

**Usage Notes**

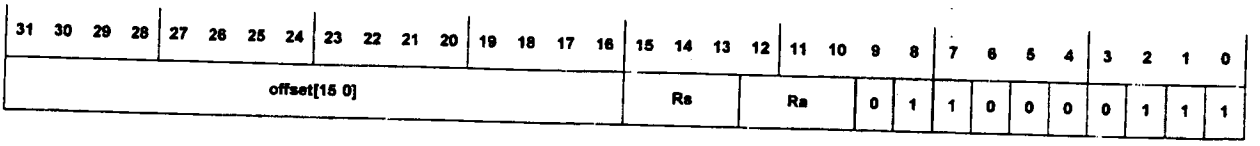
Ra and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

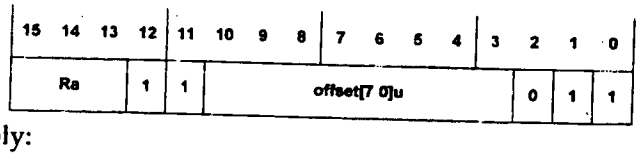
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `st.8.i %r1, @(27, %r7)`

**32-bit Encoding**




**16-bit Encoding**




The following rules apply:

- Rs is %r0.
- This encoding can represent offsets in the range 0 to 255.

# st.8.p — Store, 8-bit, PC-relative

**Instruction** `st.8.p Rs, @(offset)`

**Description** Store 8-bit value to address (offset + PC)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `(* (int8*) ( offset + PC ) ) = Rs[7:0];`

**Usage Notes**

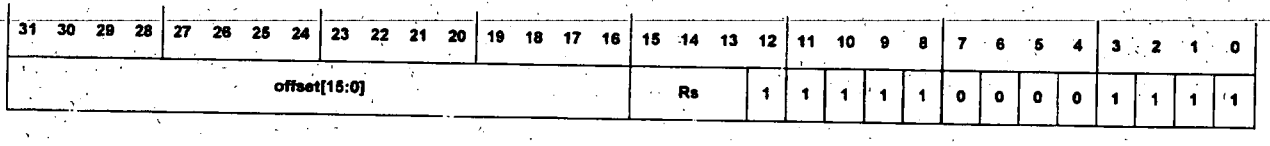
PC and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
st.8.p %r1, @(357) // Stores R1 to (357 + PC)
st.8.p %r1, @(label) // Stores R1 to 'label'
```

**32-bit Encoding**



# st.8.r — Store, 8-bit, indexed

**Instruction** `st.8.r Rs, @(Rx, Ra)`

**Description** Store 8-bit value to address (Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation**

```
* (int8*) ( Rx + Ra ) = Rs[7:0]
```

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a byte offset from Ra.

An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
st.8.r %r1, @(%r3, %r12)
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs | | | | Ra | | | | Rx | | | | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

# st.i — Store, displacement

**Instruction** `st.i Rs, @(offset, Ra)`

**Description** Store 16-bit value to address (offset + Ra)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `* (int16*) ( offset + Ra ) = Rs`

**Usage Notes**

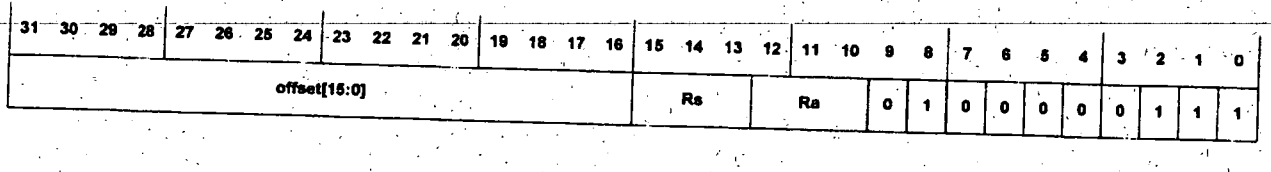
Ra and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

A 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

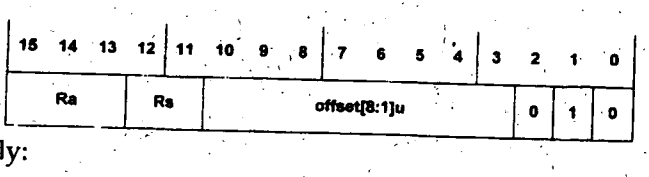
For small offsets, a 16-bit encoding of this instruction is available.

**Examples** `st.i %r1, @(27, %r7)`

**32-bit Encoding**



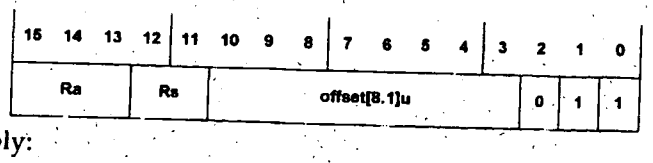

**16-bit Encoding - 1**




The following rules apply:

- The 2-bit Rs field can specify %r0, %r1, %r2 or %r3.
- This encoding can represent offsets in the range 0, 2, ..., 508, 510.

**16-bit Encoding - 2**




The following rules apply:

- The 2-bit Rs field can specify %r4, %r5 or %r6.

- This encoding can represent offsets in the range 0, 2, ..., 508, 510.

# st.p — Store, PC-relative

**Instruction** `st.p Rs, @(offset)`

**Description** Store 16-bit value to address (offset + PC)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `(* (int8*) ( offset + PC ) ) = Rs[7:0]`

**Usage Notes** PC and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

An 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
st.p %r1, @(357) // Stores R1 to (357 + PC)
st.p %r1, @(label) // Stores R1 to 'label'
```

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| offset[15:0] | | | | | | | | | | | | | | | | Rs | | | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

# st.r — Store, indexed

**Instruction** `st.r Rs, @(Rx, Ra)`

**Description** Store 16-bit value to address (2Rx + Ra)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `* (int16*) ( 2*Rx + Ra ) = Rs`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a two-byte offset from Ra.

A 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `st.r %r1, @(%r3, %r7)`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rs | | | | Ra | | | | Rx | | | | 0 | 0 | 1 | 0 | 1 | 1 | 0 |

# stz.8.i — Store zero, 8-bit, displacement

**Instruction** `stz.8.i @(offset, Ra)`

**Description** Store zero to address (offset + Ra)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
* (int8*) ( offset + Ra ) = 0x00
```

**Usage Notes**

Ra and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

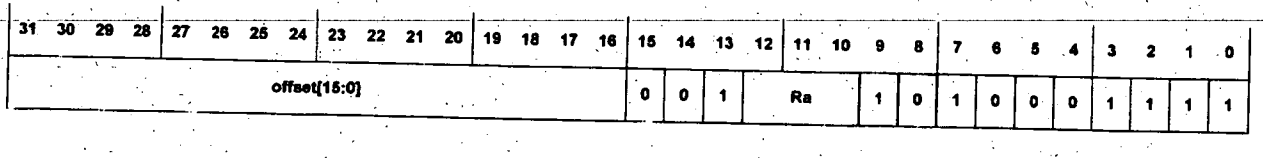
An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

For instructions where Ra is %r7, a 16-bit encoding is available.

**Examples**

```
stz.8.i @(27, %r7)
```

**32-bit Encoding**



# stz.8.p — Store zero, 8-bit, PC-relative

**Instruction** `stz.8.p @(offset)`

**Description** Store zero to address (offset + PC)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation**

```
* (int8*) ( offset + PC ) = 0x00
```

**Usage Notes**

PC and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

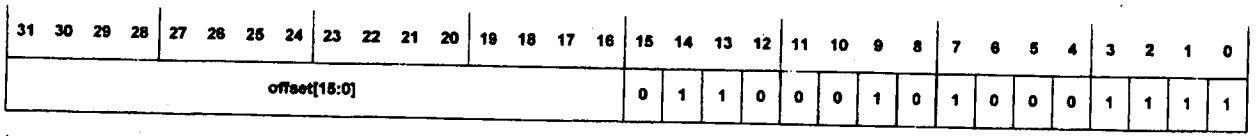
An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples**

```
stz.8.p @(27)

stz.8.p @(label)
```

**32-bit Encoding**



# stz.8.r — Store zero, 8-bit, indexed

**Instruction** `stz.8.r @(Rx, Ra)`

**Description** Store zero to address (Rx + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int8*) ( Rx + Ra ) = 0x00`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a byte offset from Ra.

An 8-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `stz.8.r @(%r1, %r7)`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | Ra | | | | | Rx | | | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# stz.i — Store zero, displacement

**Instruction** `stz.i @(offset, Ra)`

**Description** Store zero to address (offset + Ra)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int16*) ( offset + Ra ) = 0x0000`

**Usage Notes**

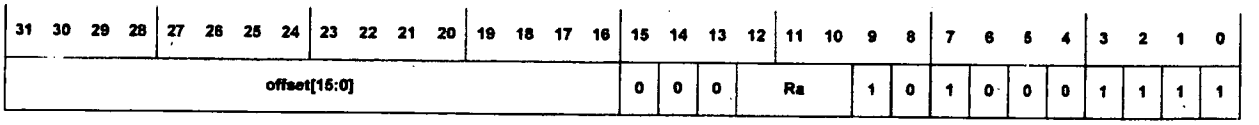
Ra and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

A 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

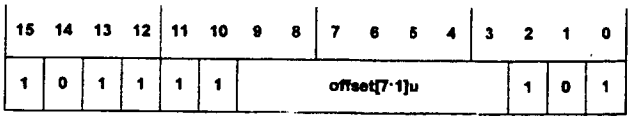
For instructions where Ra is %r7 or where the offset is small, a 16-bit encoding is available.

**Examples** `stz.i @(27, %r7)`

**32-bit Encoding**



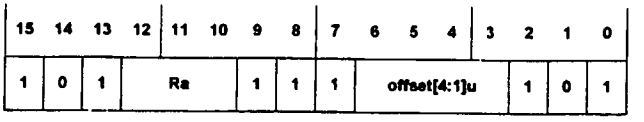

**16-bit Encoding - 1**




The following rules apply:

- Ra is %r7.
- The offset can be in the range 0, 2, ..., 252, 254.

**16-bit Encoding - 2**




The following rules apply:

- Ra is %r0 to %r6.
- The offset can be in the range 0, 2, ..., 28, 30.

# stz.p — Store zero, PC-relative

**Instruction** `stz.p @(offset)`

**Description** Store zero to address (offset + PC)

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `* (int16*) ( offset + PC ) = 0x0000`

**Usage Notes**

PC and offset are interpreted as byte addresses. The offset can take values in the range 0 to 65535 bytes and can thus address the whole of memory.

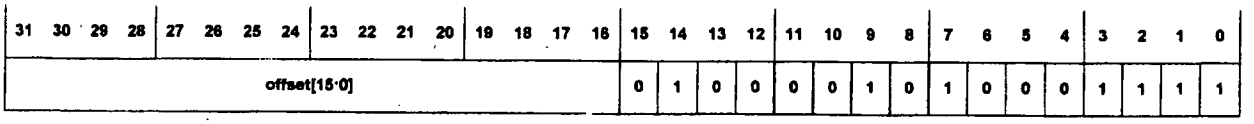
A 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

For instructions where Ra is %r7, a 16-bit encoding is available.

**Examples**

```
stz.p @(27)
stz.p @(label)
```

**32-bit Encoding**



# stz.r — Store zero, indexed

**Instruction** `stz.r @(Rx, Ra)`

**Description** Store zero to address (2Rx + Ra)

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

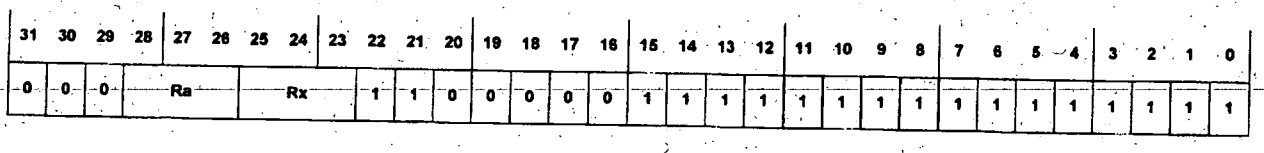
**Operation** `* (int16*) ( 2*Rx + Ra ) = 0x0000`

**Usage Notes** Ra is interpreted as a byte address. Rx is interpreted as a two-byte offset from Ra.

A 16-bit memory write is performed. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `stz.r @(%r1, %r7)`

**32-bit Encoding**



# sub.c.r — Subtract, with carry, register

**Instruction** `sub.c.r Rd, Rs, Rt`

**Description** Subtract with carry

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero and the Z flag was already set; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs - Rt - FLAGS.C`

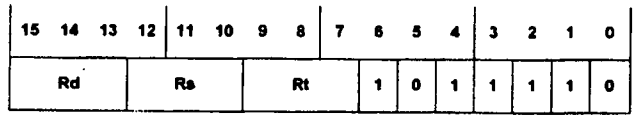
**Usage Notes** `sub.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

Rd, Rs and Rt must be in the range %r0 to %r6.

**Examples** `sub.c.r %r1, %r2, %r3`

**16-bit Encoding**



# sub.r — Subtract, register

**Instruction** `sub.r Rd, Rs, Rt`

**Description** Subtract

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| **V** | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = Rs - Rt`

**Usage Notes** `sub.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

Rd, Rs and Rt must be in the range %r0 to %r6.

**Examples**

```
sub.r %r1, %r2, %r3
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

# sub.x.i — Subtract, exchanged order, immediate

**Instruction** `sub.x.i Rd, Rs, #immediate`

**Description** Exchanged order subtract immediate

**Flags**

| | |
|---|---|
| Z | Set if the result is zero; cleared otherwise |
| N | Set if the result is negative; cleared otherwise |
| C | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| V | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = #immediate - Rs`

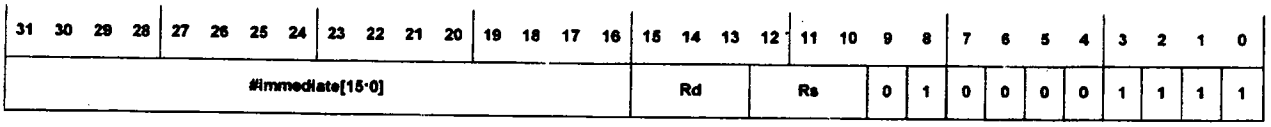
**Usage Notes** `sub.x.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

This is the same as `sub.i` but with the operand order reversed.

**Examples**

```
sub.x.i %r1, %r2, #0x1234
sub.x.i %r1, %r2, #0xFEDC
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | Rd | Rs | 0 1 0 0 0 0 | 1 1 1 1 |

# sub.xc.i Subtract, exchanged order, with carry, immediate

**Instruction** `sub.x.i Rd, Rs, #immediate`

**Description** Exchanged order subtract immediate

**Flags**

| | |
|---|---|
| Z | Set if the result is zero and the Z flag was already set; cleared otherwise |
| N | Set if the result is negative; cleared otherwise |
| C | Set if the result of the unsigned operation is incorrect; cleared otherwise |
| V | Set if the result of the signed operation is incorrect; cleared otherwise |

**Operation** `Rd = #immediate - Rs - FLAGS.C`

**Usage Notes**

`sub.xc.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

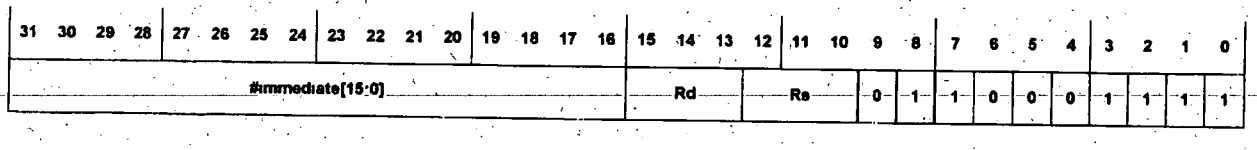
This is the same as `sub.c.i` but with the operand order reversed.

The Z flag behaviour is useful for 32-bit arithmetic.

**Examples**

```
sub.xc.i %r1, %r2, #0x1234
sub.xc.i %r1, %r2, #0xFEDC
```

**32-bit Encoding**



# swap.i — Swap register with memory

**Instruction** `swap.i Rd, @(0, Ra)`

**Description** Swap register with memory

**Flags**

| | |
|---|---|
| **Z** | Set if the new Rd value is zero; cleared otherwise |
| **N** | Set if the new Rd value is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `*Ra <-> Rd`

**Usage Notes**

The `swap.i` instruction performs an atomic swap between a register and a memory location.

Ra is interpreted as a byte address.

The memory operations are 16 bits wide. A `NullPointer` exception is thrown if the memory address is zero or within the MMU's defintion of null pointers. An `IllegalDataAccess` exception is thrown if the memory access violates the access rules implemented in the MMU.

**Examples** `swap.i %r1, (%r0, %r3)`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Ra | | | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# trap.i — Trap, immediate

**Instruction** `trap.i #immediate`

**Description** Enter Supervisor Mode from User Mode

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** -

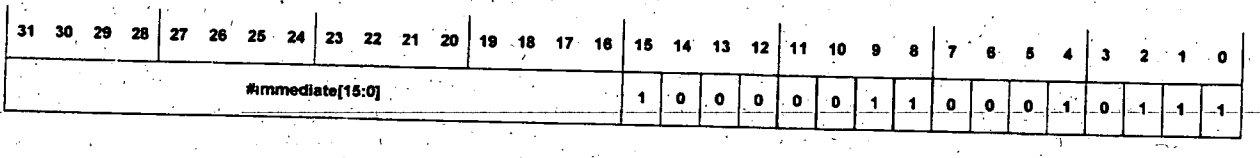
**Usage Notes** The `trap` instruction throws a `Trap` exception. This allows code executing in User Mode to call Supervisor Mode functions. The `trap` instruction is executed as a nop in the Supervisor, Interrupt and NMI modes.

The immediate can take values in the range 0 to 65535.

**Examples**

```
trap.i #0x0123
trap.i #0xFEDC
```

**32-bit Encoding**



| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| #immediate[15:0] | | | | 1 0 0 0 | 0 0 1 1 | 0 0 0 1 | 0 1 1 1 |

# trap.r — Trap, register

**Instruction** `trap.r Rs`

**Description** Enter Supervisor Mode from User Mode

**Flags**

| | |
|---|---|
| **Z** | Unchanged |
| **N** | Unchanged |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** -

**Usage Notes** The `trap` instruction throws a `Trap` exception. This allows code executing in User Mode to call Supervisor Mode functions. The `trap` instruction is executed as a `nop` in the Supervisor, Interrupt and NMI modes.

**Examples** `trap.r %r1`

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | Rs | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# ver — Read XAP4 version number

**Instruction** `ver Rd`

**Description** Read the XAP4's version number

**Flags**

| | |
|---|---|
| Z | Unchanged |
| N | Unchanged |
| C | Unchanged |
| V | Unchanged |

**Operation** `Rd = XAP4 version number`

**Usage Notes** This instruction reads the XAP4 version number.

The format of this number is:

| Bits | Meaning |
|---|---|
| 15:12 | 0000 - XAP4a<br>0001 - XAP4b |
| 11:8 | Major release number |
| 7:0 | Minor release number |

**Examples**

```
ver %r1
```

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Rd | | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

# xor.i — Exclusive-or, immediate

| | | |
|---|---|---|
| **Instruction** | | `xor.i Rd, Rs, #immediate` |
| **Description** | | Exclusive-OR with immediate |
| **Flags** | **Z** | Set if the result is positive; cleared otherwise |
| | **N** | Set if the result is negative; cleared otherwise |
| | **C** | Unchanged |
| | **V** | Unchanged |
| **Operation** | | `Rd = Rs ^ #immediate` |
| **Usage Notes** | | – |
| **Examples** | | `xor.i %r1, %r2, #0x1234` |
| | | `xor.i %r1, %r2, #0xFEDC` |

**32-bit Encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #immediate[15.0] | | | | | | | | | | | | | | | | Rd | | | | Rs | | | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

# xor.r — Exclusive-or, register

**Instruction** `xor.r Rd, Rs, Rt`

**Description** Exclusive-OR

**Flags**

| | |
|---|---|
| **Z** | Set if the result is zero; cleared otherwise |
| **N** | Set if the result is negative; cleared otherwise |
| **C** | Unchanged |
| **V** | Unchanged |

**Operation** `Rd = Rs ^ Rt`

**Usage Notes** Rd, Rs and Rt must be in the range %r0 to %r6.

**Examples** `xor.r %r1, %r2, %r3`

**16-bit Encoding**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rd | | | | Rs | | | | Rt | | | | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

# Index

*A*

Accumulators, 32-bit ..... 18
Addressing Modes ..... 40
  Displacement ..... 41
  Indexed addressing ..... 41
Aliased Instructions ..... 65
Application Models ..... 36
Assembler Syntax ..... 37

*C*

Calling Convention ..... 35

*D*

Data Alignment ..... 33

*E*

Endianness ..... 33
Exceptions ..... 22
  Break ..... 27
  DivideByZero ..... 28
  IllegalCodeAddress ..... 30
  IllegalDataAccess ..... 29
  IllegalInstruction ..... 29
  IllegalInstructionAccess ..... 29
  IllegalStackAddress ..... 30
  in the Privileged Modes ..... 23
  in User Mode ..... 23
  NullPointer ..... 28
  Returning from ..... 25
  SingleStep ..... 27
  Trap ..... 27
  UnknownInstruction ..... 28

*I*

Immediates ..... 67
  Sign extension (.s) ..... 67
  Zero extension (.u) ..... 67
Interrupt vector table ..... 26
Interrupts ..... 22, 24
  Nested ..... 25
  Returning from ..... 25

*M*

Memory Architecture ..... 21

*N*

Non-Maskable Interrupts ..... 25

*P*

Pipeline ..... 21
Processor Modes ..... 14
Processor States ..... 13

*R*

Register Lists ..... 38, 68
Registers
  Break Enable ..... 20
  Breakpoint ..... 20
  Flags ..... 19
  Normal ..... 16
  Program Counter ..... 18
  Saved Flags ..... 20
  Special ..... 18
Reset ..... 22

*S*

Stack ..... 21

*X*

XAP Family
  XAP1 ..... 8
  XAP2 ..... 8
  XAP3 ..... 8
  XAP4 ..... 6
xIDE ..... 31

-1046-

Cambridge Consultants

# XAP4a

## Hardware Reference Manual

C7432-UM-011 v1.4
24 November 2005

Cambridge Consultants

-1047-

# Revision History

| Revision | Date | Details |
|---|---|---|
| | | |
| | | |
| | | |
| | | |

# Copyright

This is the Hardware Reference Manual for the XAP4a processor. You may use this if you accept the following conditions. If you do not accept these conditions, you must delete or dispose of this copy of the XAP4a Hardware Reference Manual immediately.

Copying: You are free to copy this manual within your organisation, but you must not copy it in whole or in part to anyone outside your organisation without the prior written agreement of Cambridge Consultants Ltd

Fit for purpose: The XAP4a processor and the xIDE software development environment must not be used for safety critical and/or life support applications without a specific written agreement from Cambridge Consultants Ltd.

Liability: Cambridge Consultants Ltd. makes no warranty of any kind with regard to the information contained in this manual, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Cambridge Consultants Ltd. shall not be liable for omissions or errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material. The information contained herein may be updated from time to time.

Ownership and Licence: The XAP4a processor (including associated technologies), XAP4 documentation and the xIDE software development environment and all intellectual property rights associated with them are owned by Cambridge Consultants Ltd and are protected by copyright law and international copyright treaties, patent applications, letters patent and other intellectual property laws and treaties. You must not attempt to use the XAP4a processor or the xIDE developer's environment unless you have a valid licence agreement with Cambridge Consultants Ltd. You must not use the information within this manual or other items supplied to you for the purposes of designing, developing or testing a device or simulator which is, or is intended to be, wholly or partly instruction set compatible with the XAP4a processor. You are authorised to open, view and print any electronic copy we send you of this manual within your organisation. Printouts of this manual must be kept within your organisation.

Copyright and trademarks: This manual is © copyright Cambridge Consultants 2005 Cambridge Consultants Ltd., the Cambridge Consultants logo, XAP, XAP4, XAP4a, and xIDE are trademarks of Cambridge Consultants Ltd. All other trademarks mentioned herein are the property of their respective owners

Cambridge Consultants Ltd
Science Park
Milton Rd
Cambridge CB4 0DW
England

www.CambridgeConsultants.com/ASIC

© Cambridge Consultants Ltd 2005

Cambridge Consultants

-1048-

# Table of Contents


1 INTRODUCTION ..... 5

1.1 INTENDED READERSHIP ..... 5
1.2 ASSUMPTIONS ..... 5
1.3 ABOUT XAP4 ..... 5
1.4 THE XAP FAMILY ..... 7
1.5 XAP4A IMPLEMENTATION ..... 7
1.6 OTHER DOCUMENTATION ..... 8
1.7 FOR FURTHER HELP ..... 9
1.8 GLOSSARY AND ACRONYMS ..... 10

2 OVERVIEW ..... 11

3 GLOBAL STRATEGIES ..... 13

3.1 NAMING CONVENTIONS ..... 13
3.2 CLOCKS ..... 13
3.3 RESETS ..... 14
3.4 SYSTEM CONTROL ..... 16
3.5 ISIF INTERFACE ..... 16
3.6 TEST ..... 17

4 XAP4 CORE AND SIF ..... 18

5 MEMORY MANAGEMENT UNIT ..... 19

5.1 OVERVIEW ..... 19
5.2 MEMORY MAP ..... 20
5.3 WAIT STATES ..... 20
5.4 MEMORY EXCEPTIONS ..... 21
5.5 TIMING ..... 22
5.6 CONFIGURABILITY ..... 26

6 INTERRUPT VECTOR CONTROLLER ..... 27

6.1 OVERVIEW ..... 27
6.2 OPERATION ..... 27
6.3 INTERRUPT TIMING ..... 29

7 EXTERNAL SIGNAL DESCRIPTIONS ..... 33

7.1 TIME_STAMP ..... 33
7.2 INTERRUPTS ..... 33
7.3 XSIF INTERFACE ..... 34
7.4 ISIF INTERFACE ..... 35
7.5 CLOCKS & RESETS ..... 36
7.6 MEMORY INTERFACE ..... 38

8 INTERNAL SIGNAL DESCRIPTIONS ..... 39

8.1 INTERFACE BETWEEN XAP4_SINGLE AND XAP4_IVC ..... 39

8.2 INTERFACE BETWEEN XAP4_MMU AND XAP4_IVC ..... 40
8.3 INTERFACE BETWEEN XAP4_SINGLE AND XAP4_MMU ..... 40

**9 DEBUG SUPPORT ..... 42**

9.1 BREAKPOINTS ..... 42
9.2 SINGLE STEP ..... 43
9.3 THE SIF INSTRUCTION ..... 44

**10 SIMULATION ..... 45**

**11 SYNTHESIS ..... 46**

11.1 ASIC ..... 46
11.2 FPGA ..... 46

**12 DIRECTORY STRUCTURE ..... 47**

# 1 Introduction

## 1.1 Intended readership

This document is intended for digital designers who wish to integrate the XAP4a processor and associated support functions into an ASIC or FPGA. It provides general overview information and interface signal descriptions.

## 1.2 Assumptions

The reader is assumed to be familiar with microprocessor architectures in general, and to have some understanding of digital electronics.

## 1.3 About XAP4

The XAP4 is a powerful 16-bit processor optimised for low gate count and low power. It has a von Neumann architecture and can address 64KByte of linear byte-addressable memory.

### *High Code Density*

The XAP4 architecture, instruction set, compiler toolchain and application binary interface are all designed for efficient execution of programs written in C.

### *Complex Systems*

The XAP4 implements the features necessary to support complex software systems where high reliability or high integrity is a requirement. The XAP4 provides hardware support for common real time operating system primitives, including a clear definition of user and supervisor modes, atomic instructions for synchronisation constructs, and position independent code to allow dynamic linkage.

The XAP4 MMU interface enables memory protection systems for instruction and data, throwing exceptions if processes access code or data memory illegally.

### Interrupts and Exceptions

The XAP4 supports a total of 16 interrupts and 16 exceptions. Each has its own handler defined in a vector table.

### Rich RISC-like Instruction Set

A rich set of instructions is provided, including a complete set of branches, arithmetic operations on 8, 16 and 32 bit data. Several instructions map closely onto C language constructs thereby providing very high code density and fast execution.

### 16-bit and 32-bit Instructions

The XAP4 instruction set contains 145 unique instructions. Most common instructions are 16 bits long with less common instructions using 32 bits. Some instructions have 16-bit and 32-bit forms, with the toolchain using the 16-bit form whenever possible.

This approach gives high code density. Programs can mix 16-bit and 32-bit instructions at will. No switching between modes is necessary and the processor executes all instructions at full speed. The 16-bit instructions can access all of the processor resources available to the 32-bit instructions.

### Hardware Acceleration

The XAP4 processor includes a hardware multiplier, divider and barrel shifter. Instruction execution is pipelined, resulting in operating speeds of over 100MHz on a 0.18µm ASIC process. The processor can be implemented in any ASIC process between 2.0µm and 0.065µm and in any FPGA.

### Targeted for Speed or Size

The XAP4 instruction set architecture is available in different hardware implementations. The XAP4a implementation is targeted for low cost, low power applications and requires approximately 9k gates for the processor core.

### Language Support

The GNU Compiler Collection (gcc) provides an industry standard ANSI C compiler plus GCC extensions. xap4-gcc and binutils (assembler and linker) are provided in the XAP4 software toolkit.

### Development Tools

The xIDE integrated development environment provides a cross-platform development and debugging tool for XAP4 programmers, using an industry-standard look and feel.

## 1.4 The XAP Family

The XAP4 is the smallest member of Cambridge Consultants' new Von Neumann architecture XAP processor family :

- XAP4 16-bit processor supporting up to 64KB of memory, 9k gates
- XAP5 16-bit processor supporting up to 16MB of memory, 12k gates
- XAP3 32-bit processor supporting up to 4GB of memory, 30k gates

These processors have all been designed for embedded use in ASICs and FPGAs. They all use Cambridge Consultants' patented SIF interface for debugging and the xIDE development environment for software development. They all have excellent code density when used with their GCC/binutils compile chains. XAP3 also has an ANSI C compiler that provides even higher code-density. They all have 4 processor modes with excellent support for RTOS and low-latency interrupts. They all have the same style of instruction set and programmer's model. The family offers an easy upgrade path between systems of different sizes, offering the programmer a consistent interface.

The new XAP family builds on the success of Cambridge Consultants legacy 16-bit Harvard architecture XAP processor family :

- XAP1 Up to 144KB Program memory and 128KB Data memory, 3k gates
- XAP2 Up to 32MB Program memory and 128KB Data memory, 12k gates

XAP1 has been used in many ASIC projects since its development in 1994, resulting in over 10 million silicon devices.

The XAP2 is an evolutionary development from XAP1. XAP2 has been used in over 100 million silicon devices since its development in 1999. This includes all of CSR's Bluetooth and 802.11 devices.

## 1.5 XAP4a Implementation

The XAP4 instruction set architecture is available in different hardware implementations.

The XAP4a hardware implementation exists as a soft IP core written entirely in synthesisable Verilog RTL. This can be targeted at either System-on-Chip ASIC or FPGA implementations.

The xIDE toolset includes an instruction set simulator for XAP4a. This can be extended to include models of other parts of the system, including memories and interrupts.

The Verilog design observes the following conventions:

- The XAP4a system does NOT contain any latches.
- The XAP4a system itself does not contain any memory. The user is responsible for connecting external memory to the design. The XAP4a system is supplied with examples of synchronous memory. For optimal performance it is recommended to have 16-bit wide memory with independent byte write enable capabilities, and single-cycle access.
- The XAP4a system itself does not contain any I/O registers, neither does it have specific instruction to perform I/O operations. All I/O must be memory mapped and are accessed using the normal XAP4 instructions. As with the memories the I/O register should be synchronous. The XAP4a system comes with examples of synchronous, memory mapped I/O registers.
- The XAP4a system itself does not contain scan or any other logic for testing, However there should be no problems using it with a standard test compiler as the design is single edge triggered, has a fully asynchronous reset and contains no latches.
- All state machines are fully decoded.

## 1.6 Other Documentation

For details of the XAP4a hardware, the following documents are available :

| Document | Contains |
|---|---|
| XAP4a Hardware Quickstart Guide<br>C7432-UM-001 | A short tour of the XAP4a hardware release. Intended to get users up and running quickly with Verilog simulation, ASIC synthesis and FPGA synthesis. |
| C7432-UM-011 | XAP4a Hardware Reference Manual (this document) |

For details of the SIF debug environment, the following documents are available :

| Document | Contains |
|---|---|
| C4862-TM-023 | User SIF Definition |

For details of the XAP4 software development tools, the following documents are available:

| Document | Contains |
|---|---|
| xIDE User Manual C7066-TM-001 | Describes the general features of xIDE. |
| xIDE Python Object Model C7066-TM-003 | Describes the python object model used within xIDE. This is essential for developers wanting to use the scripting features of xIDE. This allows developers to perform automated testing or to model other parts of a system during simulation. |
| xIDE for XAP4 C7066-TM-017 | Describes specific features of the XAP4 plugin for xIDE. |
| XAP4 Programmer's Manual C7432-UM-002 | Provides the primary reference for software engineers developing programs to run on XAP4 systems. It contains all the information needed to understand the XAP4 architecture from a software perspective. |
| XAP4 BinUtils Manual C7432-UM-003 | Primary reference for software engineers using the XAP4 binutils assembler and linker. It references GNU documents available on the internet. |
| XAP4 GCC Compiler Manual C7432-UM-004 | Primary reference for software engineers using the xap4-gcc compiler. It references GNU documents available on the internet. |
| XAP4 Instruction Set Quick Reference Card C7432-UM-005 | Summary behaviour of all XAP4 instructions.<br>Indicates the number of clk cycles required in XAP4a implementation. |

For details of Cambridge Consultants Silicon IP range, refer to :

| Document | Contains |
|---|---|
| ASICs-SB-001 | Overview of Silicon IP available from Cambridge Consultants |
| XAP4 Datasheet ASICs-SB-011 | Short overview of the XAP4 architecture. |

## 1.7 For Further Help

The XAP4 documentation set provides answers to most questions.

If a problem remains unsolved, contact Cambridge Consultants technical support at:

xap@CambridgeConsultants.com

## 1.8 Glossary and Acronyms

| Term | Meaning |
|---|---|
| Double-Word | A 4-byte (or 32-bit) quantity. |
| Word | A 2-byte (or 16-bit) quantity. The XAP4 is a 16-bit processor. |
| Byte | An 8-bit quantity. |
| Little Endian | A system in which the least significant byte of a word is stored at the lowest memory address. The XAP4 is a little endian processor. |
| Big Endian | A system in which the most significant byte of a word is stored at the lowest memory address. |
| iSIF | Internal Serial InterFace. |
| xSIF | External Serial InterFace. |
| ALU | Arithmetic and Logic Unit. |
| IVC | Interrupt Vector Controller. |
| IVT | Interrupt Vector Table. |
| MMU | Memory Management Unit. |
| NMI | Non-Maskable Interrupt. |
| RTL | Register Transfer Level |
| Verilog | The hardware description language used for XAP4 |

# 2 Overview

A typical ASIC containing a XAP4a system is shown in the figure below. This illustrates the xap4_system module connected to application-specific circuitry to provide:

- System control - clock and reset generation, watchdog functions, etc.
- Memories, peripherals and interrupt sources.
- An optional time-stamp module (timer) that provides a time-stamp on all SIF transfers.
- An optional internal SIF (iSIF) interface

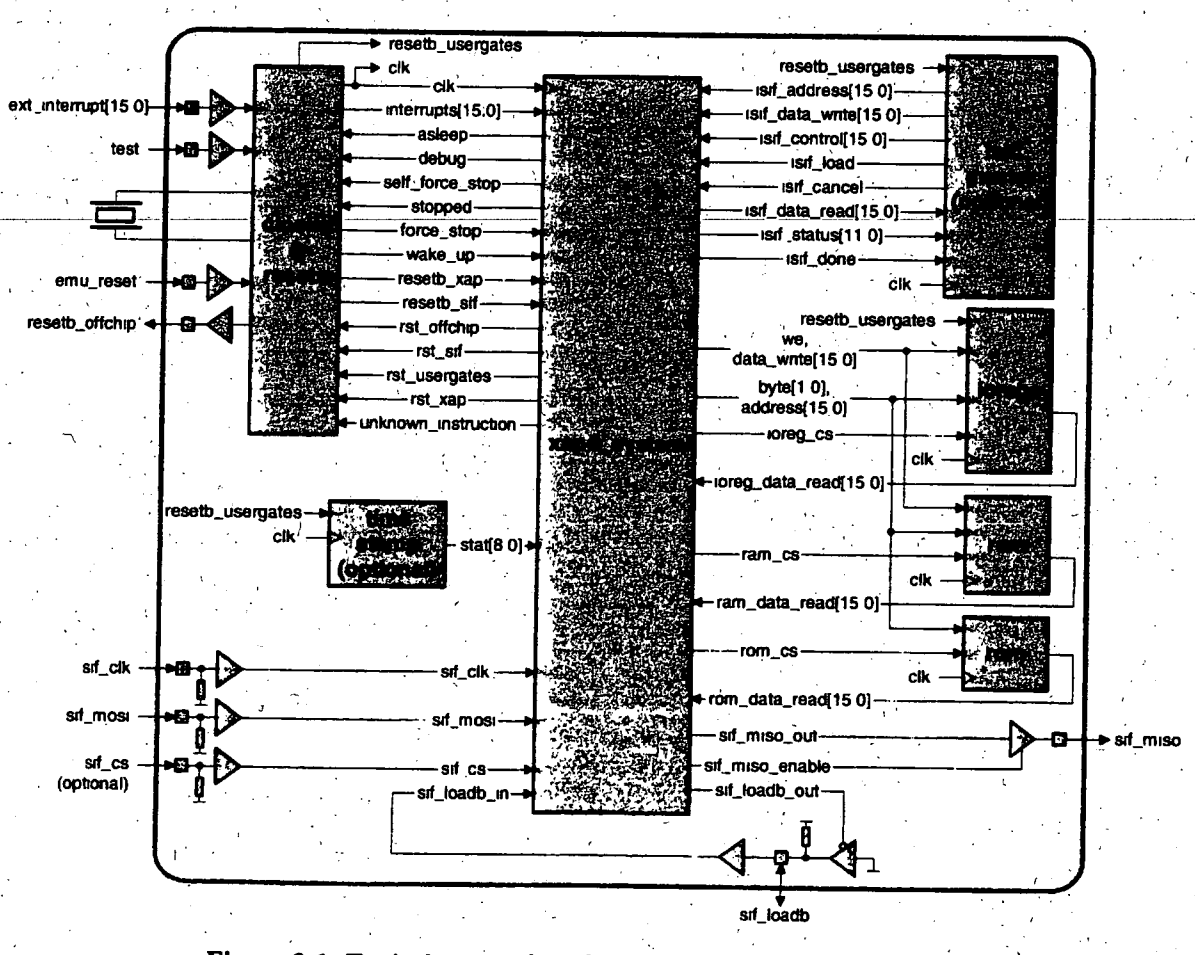



**Figure 2-1: Typical connections for XAP4a within an ASIC**

The figure below shows the typical structure within the xap4_system module. Normally, this contains three modules: xap4_single, xap4_mmu, and xap4_ivc.

- xap4_single: this contains a single instance of the xap4a processor, and a SIF interface for debug and development support. This module should be used as delivered from Cambridge Consultants. The customer is not expected to modify any part of xap4_single.

- xap4_mmu: this is a Memory Management Unit (MMU) which interfaces with memory-mapped I/O registers and memories, typically ROM and RAM. The xap4_mmu delivered is an example. The customer is expected to modify this as required for his particular ASIC or FPGA.

- xap4_ivc: this is an Interrupt Vector Controller (IVC) that prioritises interrupt sources and provides an interrupt number to the XAP4. The xap4_ivc delivered is an example. The customer is expected to modify this as required for his particular ASIC or FPGA.

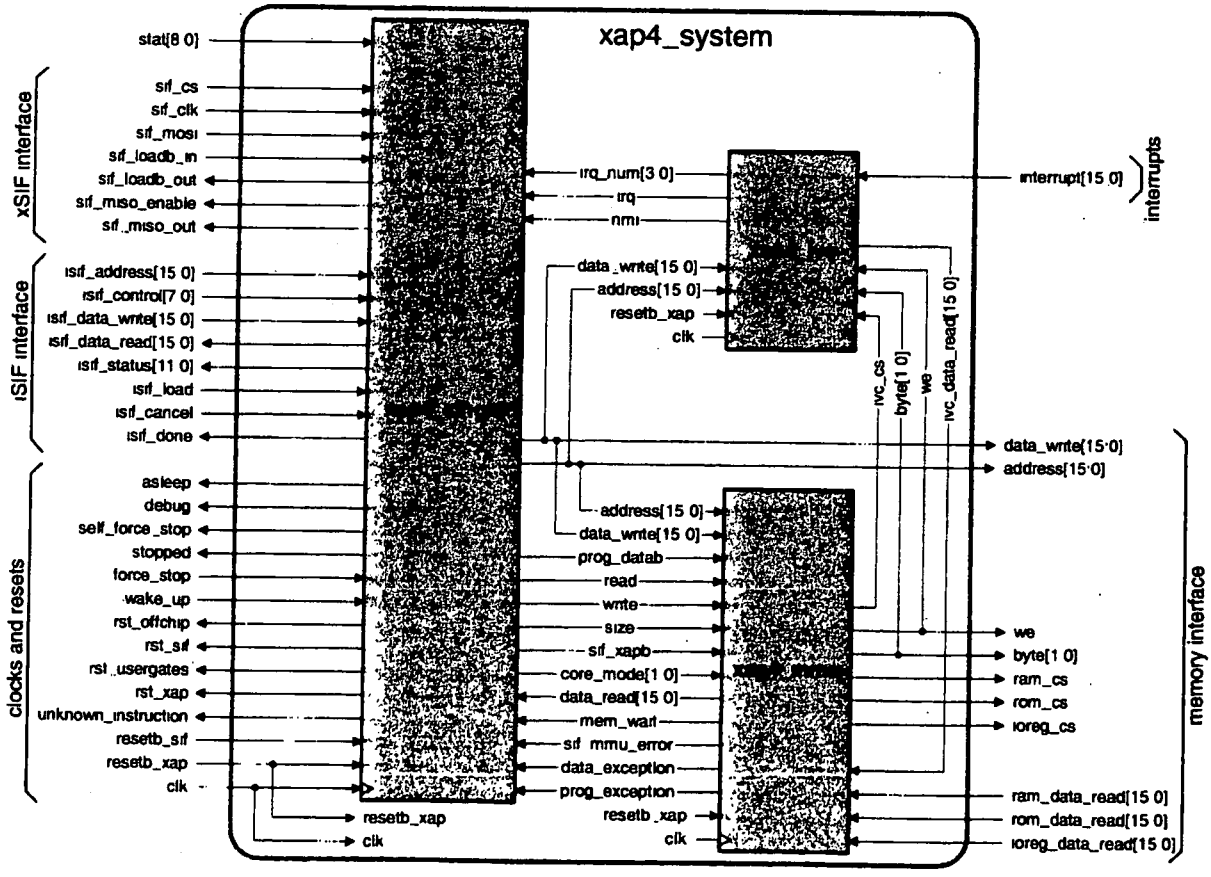



**Figure 2-2: Typical XAP4a System**

# 3 Global Strategies

## 3.1 Naming Conventions

All Verilog names (modules, ports, signals) are lower case. Words are separated with an underscore character. Buses are always of the form [high:low]. Module names all start with 'xap4_'. Control signals are normally active high. If active low, the signal name (or intermediate word) ends in b. e.g. :

- `resetb_xap`
- `prog_datab`
- `xap4_single`
- `address[15:0]`
- `size`

The high level modules have been created as Fletcher charts that automatically generate the Verilog RTL to be used. This methodology uses a flat-naming convention. This means that a signal has the same name at all levels of the hierarchy. This means that signals and ports have the same names.

The low level modules are hand written in Verilog RTL. In general this maintains the flat naming strategy, but it may not always be the case.

## 3.2 Clocks

There are no latches or memories inside xap4_system. The only sequential components used are clocked flip-flops.

There are only 2 clocks used in xap4_system. Inside xap4_system, the 2 clocks are only used to clock flip-flops.

| Name | Description | Used |
|---|---|---|
| clk | Main system clock | Most flip-flops in xap4_system are clocked by clk. These are all clocked on posedge clk.<br><br>clk is normally the system clock used for the RAM, ROM and other digital logic outside xap4_system. |
| sif_clk | SIF clock | Used for SIF shift register. Not expected to use this signal for anything outside xap4_system.<br><br>Different parts of the SIF shift register use posedge and negedge of sif_clk. |

For the following XAP4 modules' boundaries :

- xap4
- xap4_xsif
- xap4_ivc
- xap4_mmu

The XAP4 input sif_mosi is sampled on negedge sif_clk. All other XAP4 inputs are sampled on posedge clk. Some of these inputs go directly into the flipflop D-pin. Other inputs have combinational logic between the module input and the flipflop D-pin.

The XAP4 output sif_miso_out changes on posedge sif_clk. All other XAP4 outputs change on posedge clk. Some of these outputs come directly from the flipflop Q-pin. Other outputs have combinational logic between the flipflop Q-pin and the module output. Such signals may bounce for a short period after posedge clk before assuming their new stable value.

Clocks are not gated at all within xap4_system. If the ASIC designer wishes to gate clocks, (for example when the XAP4 indicates that it is in a sleep state), this should be done externally. An example of how to do this is provided in the clocks_and_resets module supplied by Cambridge Consultants.

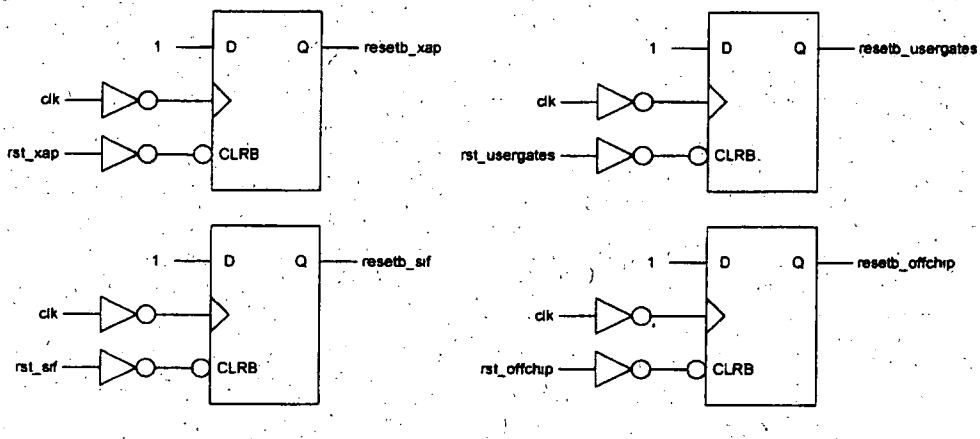
## 3.3 Resets

We expect the ASIC designer to create a single module called clocks_resets for the whole ASIC. All clock, reset and test signals should either pass through or be

generated from within the clocks_resets module. This allows the designer to have a single block to analyse for the correct operation of clocks and resets for the ASIC in functional and in all test modes (including scan path).

We expect the clocks_resets module to generate 4 resetb signals. resetb_xap and resetb_sif are fed to xap4_system. resetb_usergates may be used to reset another on-chip region. resetb_offchip may be used to reset off-chip devices. These signals should be generated from the associated 4 rst outputs from xap4_system. They may also be generated by other reset sources such as power-on-reset or watchdog.

The 2 resetb inputs to xap4_system go to the asynchronous clrb or setb pins of all flip-flops inside xap4_system. These signals are all active low. The clocks_resets module which generates the resetb signals should synchronise the end of each reset (i.e. posedge resetb) with negedge clk. A possible circuit is shown below.




The 4 rst signals are all outputs from flip-flops in xap4_system clocked on posedge clk. These signals are all active high and are one clock cycle wide. All 4 can be generated in various combinations by SIF commands from the debug computer. These can be connected in hardware (via clocks_resets module) to reset different sections of the ASIC. The ASIC designer is free to connect up some or all of these rst outputs as best supports his system reset strategy. It is only mandatory to connect the rst_xap and rst_sif signals. See the example RTL in xap4_clocks_resets for guidance.

The `unknown_instruction` output is also clocked on posedge clk. It is generated by the xap4 when it encounters an unrecognised opcode and it is not in User mode. (An unknown instruction in User mode generates an exception that can be processed by an exception handler.) An unknown instruction in any privileged mode (Supervisor, Interrupt, NMI) is typically considered a fatal error. We expect this signal to be used as input to the clocks_resets module, so that it can be used to set the various `resetb_signals` high.

| Name | Description |
|---|---|
| resetb_sif | Resets all flip-flops in xap4_xsif and the debug section of xap4. The xap4 is reset to normal mode and run continuous. |
| resetb_xap | Resets all other flip-flops in xap4, xap4_ivc, xap4_mmu. The SIF mode (normal or debug) is not altered, nor is the run state of the XAP4 (run continuous, run to break, single-step, or stop). |
| rst_offchip | May be connected via an ASIC pad to reset off-chip devices |
| rst_usergates | May be connected to reset a region of User gates (excluding the XAP4/SIF). |
| rst_xap | Should be connected to reset the XAP4 via resetb_xap. |
| rst_sif | Should be connected to reset the SIF via resetb_sif. |

## 3.4 System Control

XAP4 has a number of inputs and outputs for overall system control. The four output signals are asleep, debug, self_force_stop, and stopped. These are all synchronous signals, clocked on posedge of clk. The two input controls are wake_up and force_stop, which are synchronously sampled by XAP4 on posedge of clk. Both these inputs should be tied low if not used.

Further description for all these signals is provided in section 7.5.

## 3.5 iSIF Interface

The ASIC block diagram in Figure 2-1 shows an optional isif_master module. This uses the iSIF (internal SIF) interface to provide powerful debugging and control capabilities from another processor or control logic within the ASIC. The iSIF is expected to be used in one of the three following ways:

- Unused. Tie all outputs from isif_master to all zeros. This will ensure that almost all the iSIF related logic within XAP4 is optimised away during synthesis.
- Memory access only, such as program download or data acquisition. The xSIF interface (if connected) would continue to provide debugging capability.

- Memory access as above, plus debugging capability. The xSIF interface (if connected) would not be used for debugging.

## 3.6 Test

There are no special features in XAP4a for test. It has been designed for simple scan-path testing. It provides the designer with clean access to the clock and reset signals. It is recommended that any special test mode behaviour is implemented in the clocks_resets module. This may modify clock and reset behaviour in the various test modes.

# 4 XAP4 core and SIF

The xap4_single module provides the functionality necessary to implement the XAP4 instruction set. It decodes the instruction fetched from memory, and then executes the necessary ALU operation(s) and/or memory load operations, writing the appropriate results back to the internal registers, flags, and memory as necessary. The core contains the following features:

- Single cycle for most instructions. The core utilises a two-stage pipeline, fetching the next instruction while decoding and executing the present instruction.
- The ALU provides 4 groups of functions: add/subtract, multiply/divide, shift/rotate, and logic operations (and, or, xor, etc.)
- Register file for the r0-r7 processor registers, with one write port and three read ports. The registers are split into even and odd groups, allowing a pair of contiguous registers to be read or written in one cycle (for 32-bit data).
- Fully integrated support for SIF read/write access to memory, xap4 registers and debug control (breakpoint, single-step, etc.), allowing the system to be debugged with the xIDE integrated development environment. The ASIC is the SIF Slave. The external SIF Pod is the SIF Master.

# 5 Memory Management Unit

## 5.1 Overview

The MMU module forms the interface between xap4_single and external memories and peripherals. The XAP4a is provided with an example MMU (xap4_mmu) but should be adapted to the user's specific requirements. The MMU provides the following functions:

- Decoding of the address to provide chip-select signals to the memory devices.
- Multiplexing of the read data from the memory devices.
- Support for slow/busy memory devices, using a `mem_wait` signal.
- Detection and reporting of memory access exceptions.

The xap4_single module initiates a memory access by setting high either `read` (for a read access) or `write` (for a write access). It also drives `address[15:0]` and `size` to indicate the target address and the width of the access (1 or 2 bytes). If this is a write access, it also drives `data_write[15:0]`.

The MMU decodes the address, and determines first if the access is to be permitted or whether it should cause an exception. If the access is not permitted, the MMU flags an exception (see section 5.4). Otherwise, the MMU sets high the appropriate chip select signal (e.g. `ram_cs` for a write to RAM), and sets `we` high for a write access or low for a read access. It also and sets `byte[1:0]` according to `address[0]` and `size`:

| Access type | `size` | `address[0]` | `byte[1:0]` |
|---|---|---|---|
| Access to data bits [7:0] | 0 | 0 | 00 |
| Access to data bits [15:8] | 0 | 1 | 01 |
| Access to data bits [15:0] | 1 | x | 11 |

-1065-

## 5.2 Memory Map

The MMU decodes the incoming address from the XAP to create one or more chip-select signals to the various memory devices. The figures below show the recommended memory maps for the four most common configurations.

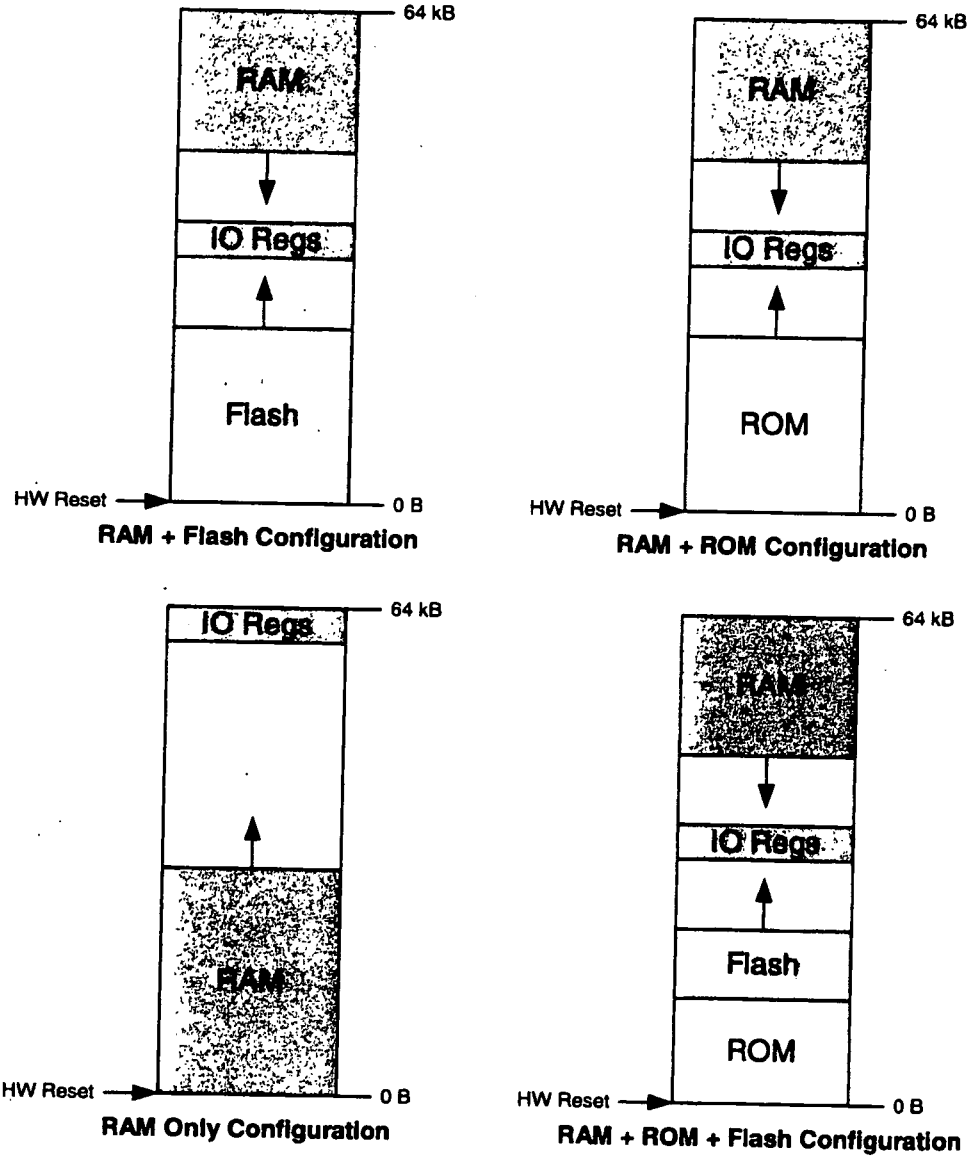



**Figure 5-1: Recommended Memory Maps**

## 5.3 Wait States

The MMU is responsible for informing the xap4 processor if wait states are required. The MMU drives `mem_wait` high unless this is the selected memory can accept a

new address on the next clock cycle. The xap4_single module holds its memory interface outputs stable until `mem_wait` goes low.

When xap4_single samples `mem_wait` low at a clock edge, it is free to begin a new memory access. Note that for a read access, the read data is returned on the clock *after* the one in which `mem_wait` goes low – see the memory read timing described in section 5.5.3 below.

For best performance, the xap4 should be connected to memory (data and program) that operates with zero wait states, i.e. single cycle memory. This means that the memory should be ready to accept a new address for a read or write access on every clock cycle. Since most xap4 instructions make full use of the memory bandwidth, the use of memory with one or more wait states will degrade xap4 performance almost linearly.

The example MMU defines memory with the following performance:

- RAM: zero wait states (one cycle)
- ROM: one wait state (two cycles).
- I/O space: zero wait states (one cycle)

The user can modify the xap4_mmu module to suit the memory and peripherals connected to the MMU. If memory with one or wait states is to be used, then the user should refer to section 9.3 that discusses how this impacts the use of the XAP4 sif instruction.

## 5.4 Memory Exceptions

The MMU has two memory exception flags that it drives to the xap4: `data_exception` and `prog_exception`. These can be used to notify the xap4 of illegal data or program accesses respectively. In addition, the MMU outputs an error flag, `sif_mmu_error`, that it can set or clear during SIF accesses to memory. This signal can be used by the MMU to notify the SIF interface of an illegal memory access

The MMU receives various qualifiers from the xap4_single module to assist it in qualifying memory access exceptions:

- `prog_datab` – to indicate whether xap4 accesses are for data or program,
- `sif_xapb` – to indicate the source of the write access (sif or xap4),
- `core_mode[1:0]` – to indicate the current mode of the xap4 processor

In the example MMU provided, exceptions are generated as follows:

- `prog_exception` is set high for a program access to I/O register space, MMU register space, or IVC register space.
- `data_exception` is set high for a data write to ROM space or for a data read/write to MMU register space.
- `sif_mmu_error` is always set low (i.e. the SIF is permitted full access to the entire memory space).

It is a straightforward task for the user to modify the MMU to extend exception detection as required. For example, it may be desired to prevent program access to certain areas of RAM in User mode, or to prevent data access to portions of RAM, or to flag byte writes to memory without byte-enables, or to differentiate between SIF access and xap4 access to memory. These and other combinations can be achieved with the signals provided to the MMU.

## 5.5 Timing

### 5.5.1 Memory access generating an exception

The following figure illustrates an access to memory that generates an exception. The xap4 processor begins the access (a program read in this example) at clock edge c1. The MMU decides that this is an invalid access, and drives `mem_wait` low in the same cycle. No chip selects are set high, so no memory access occurs. The MMU signals that this was a invalid program access by driving `prog_exception` high for the xap4 processor to sample on the next clock edge, c3. If this were a data access that the MMU detected as invalid (i.e. if prog_datab is low), then the MMU would set high `data_exception` instead of `prog_exception`.

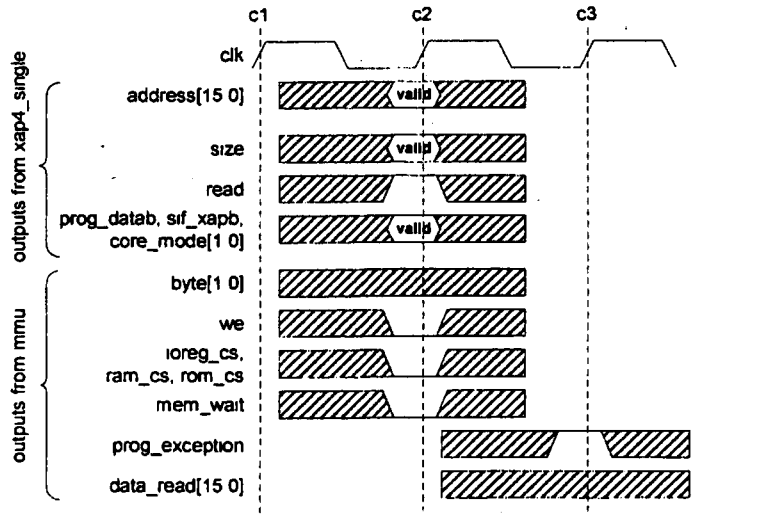



**Figure 5-2: Program read from memory, exception generated**

### 5.5.2 Memory Write

The following figure illustrates a write access to memory with zero wait states. The xap4 processor begins a write access at clock edge c1. The MMU decides that this is a valid write access requiring no wait states, and drives `mem_wait` and its outputs to the memory in the same cycle. Thus all inputs to the memory are valid at edge c2. The MMU signals that this was a valid access by driving `data_exception` and `prog_exception` both low for the xap4 processor to sample on the next clock edge, c3.

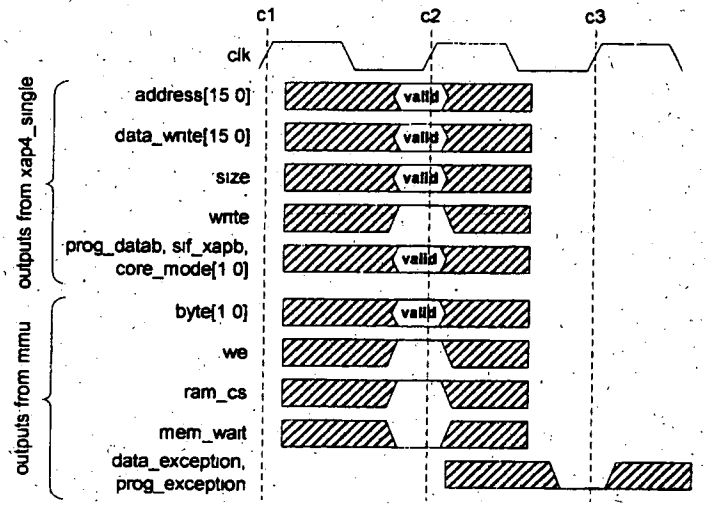



**Figure 5-3: Valid write to memory, zero wait-states**

The following figure illustrates a memory write access requiring one wait state. Note that the MMU drives `mem_wait` high for clock edge c2, and then low for clock edge c3. Note also that the xap4 _single and the mmu module both maintain their outputs to the memory through to clock edge c3.

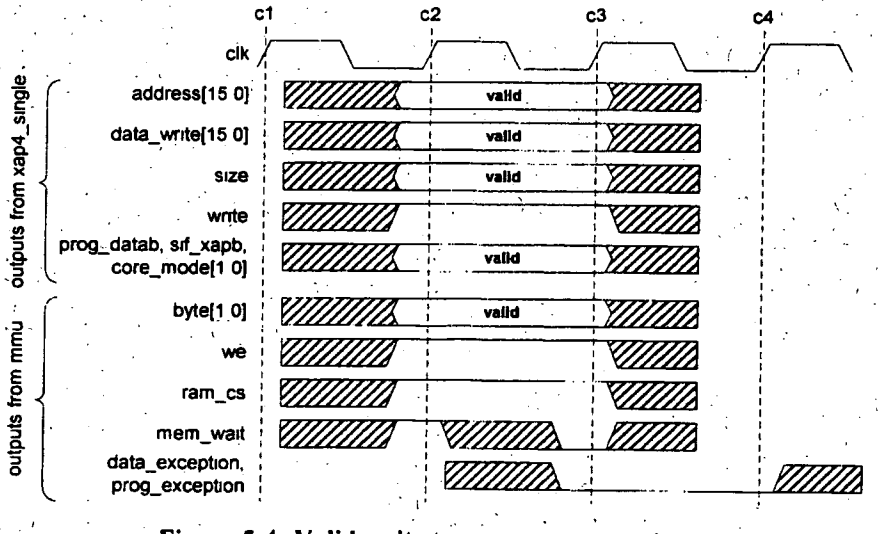



**Figure 5-4: Valid write to memory, one wait-state**

### 5.5.3 Memory Read

The following figure illustrates a read access to memory with zero wait states. The xap4 processor begins a read access at clock edge c1. The MMU decides that this is a valid read access requiring no wait states, and drives `mem_wait` and its outputs to the memory in the same cycle. Thus all inputs to the memory are valid at edge c2. The MMU signals that this was a valid access by driving `data_exception` and `prog_exception` both low for the xap4 processor to sample on the next clock edge, c3. The MMU selects the appropriate read data (e.g. `ram_data_read[15:0]` for a read from RAM) and passes this through as `data_read[15:0]` in time for the xap4 processor to capture at clock edge c3.

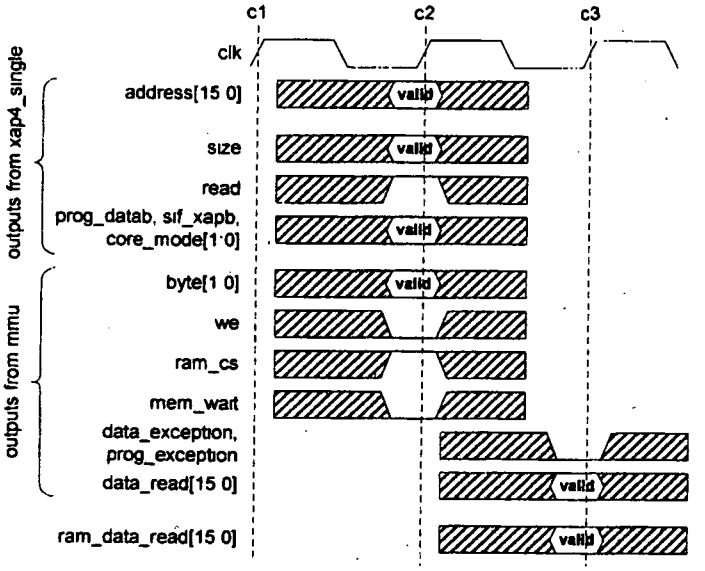



**Figure 5-5: Valid read from memory, zero wait-states**

The following figure illustrates a memory read access requiring one wait state. Note that the MMU drives `mem_wait` high for clock edge c2, and then low for clock edge c3. Note also that the xap4 _single and the mmu module both maintain their outputs to the memory through to clock edge c3. The MMU drives `data_read[15:0]` in time for the xap4 processor to capture at clock edge c4.

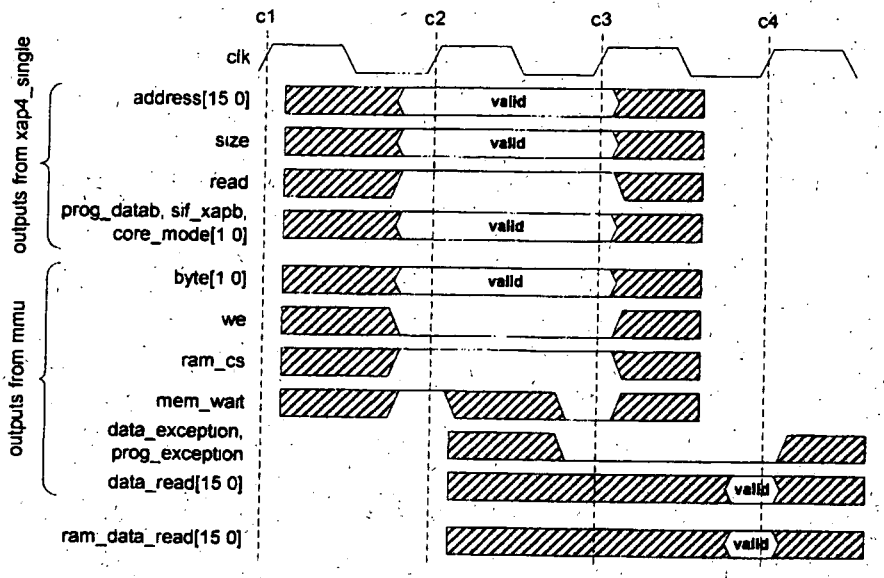



**Figure 5-6: Valid read from memory, one wait-state**

### 5.5.4 Sequential memory accesses

In normal operation, accesses to memory will be sequential. The xap4 system design provides pipelined operation for maximum throughput. Total throughput is one clock per access for zero wait state memory, two clocks per access for one wait state, etc. The following figure illustrates a sequence comprising:

- Read RAM, zero wait states
- Write RAM, zero wait states
- Read I/O registers, one wait state

These are colour-coded in the figure to assist the reader in correlating signal activity with the respective memory accesses.

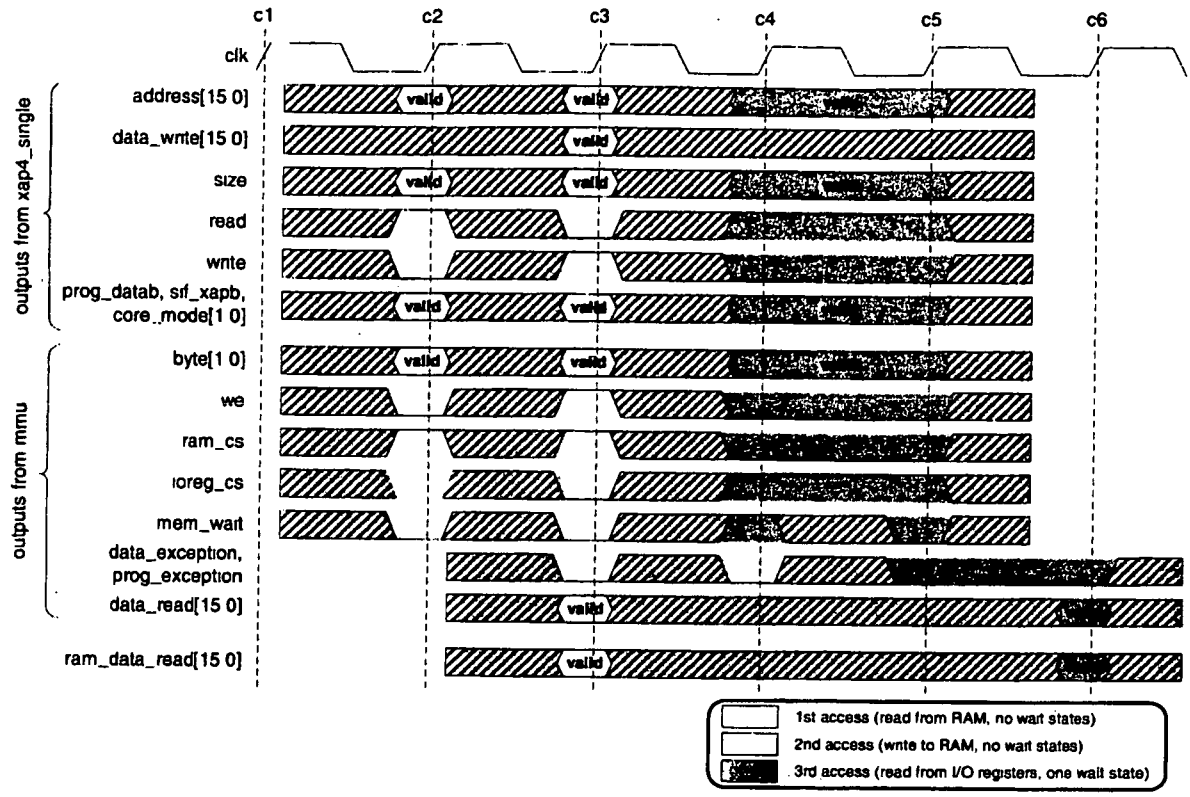



**Figure 5-7: Sequential memory accesses**

## 5.6 Configurability

This chapter has discussed various capabilities provided by the MMU, including address decoding, exception detection, and wait-state control. For most users, these capabilities will be fixed and can be coded directly into the RTL. However, some users may require that some or all of these capabilities be software-configurable. This can be achieved by modifying the MMU to include I/O mapped registers within the MMU to specify these parameters. For example, a pair of registers could be added to specify the base address and size for RAM, or a set of registers to specify the address range for which data access is permitted. Alternatively, some users may wish to add registers that capture certain signals when memory exceptions are detected.

These registers are not included in the example MMU, since the need for them and their precise format will be highly application dependent. In addition, their inclusion would unnecessarily increase chip area and power for those users that do not need this capability. However, it is relatively straightforward to add these registers if needed, and the example MMU does provide address decoding to access these registers.

# 6 Interrupt Vector Controller

## 6.1 Overview

The IVC module forms the interface between xap4_single and all external interrupts. The XAP4a is provided with an example IVC (xap4_ivc) but should be adapted to the user's specific requirements.

The Interrupt Vector Controller (IVC) accepts up to 16 interrupts, named interrupt[15:0]. The IVC captures incoming interrupt pulses, and then generates an interrupt and interrupt number to xap4_single. The interrupt priority scheme can be customised to meet the user's requirements.

## 6.2 Operation

### 6.2.1 Registers

The example IVC includes three 16-bit registers:

- ivc_status register (read-only)
- ivc_clear register (write-only; read always returns all zeros)
- ivc_enable register (read/write)

The functions of these three registers are explained in the following paragraphs.

### 6.2.2 Interrupt capture & clear

The IVC accepts 16 external interrupts (`interrupt[15:0]`). It assumes that these inputs have already been synchronised and converted to high-going pulses, one clock wide. (This is done in the example clocks_and_resets module provided by Cambridge Consultants.)

The example IVC sets the corresponding bit in the ivc_status register if a pulse is detected on `interrupt[7:0]`. If more than 8 interrupts are required, it is a simple matter to modify the IVC to use the remaining 8. If more than 16 interrupts are required, this can be achieved by grouping several interrupts to share a single

interrupt number. The interrupt handler software would then need to read additional registers in the IVC to determine and clear the precise source of the interrupt.

Software can read the ivc_status register at any time to determine which interrupts have been detected since the last time any have been cleared. Software can clear one or more bits in the ivc_status register by setting the corresponding bit(s) in the ivc_clear register. For each bit set high during a write to the ivc_clear register, the IVC will reset to zero the corresponding bit in the ivc_status register. For each bit set low during a write to the ivc_clear register, the corresponding bit in the ivc_status register will be unaffected.

Note that if a new interrupt arrives on the same clock cycle as it is being cleared by software, the arriving interrupt takes precedence and will set the corresponding bit in the ivc_status register. This ensures that an incoming interrupt cannot be inadvertently 'lost'.

### 6.2.3 Interrupt enable

Each captured interrupt is then individually 'AND'-ed with its corresponding bit in the ivc_enable register. The ivc_enable register allows software to control which interrupt sources are allowed to generate an interrupt to the xap4. If a bit in the ivc_enable register is set high the corresponding interrupt is enabled, otherwise it is disabled.

### 6.2.4 Priority encoding

All enabled interrupts are then passed to a priority encoder. This sets the `irq` signal to the xap4 if any of the enabled interrupts are active, and also sets `irq_num[3:0]` to indicate the number of the highest priority active interrupt. The example IVC implements a straightforward fixed priority scheme, with interrupt[7] being the highest priority and interrupt[0] being the lowest. This can of course be modified if a more complex priority scheme is required, such as a round-robin or a hybrid scheme.

When xap4_single detects `irq` set high, and it is in the appropriate mode (any mode except NMI Mode), it will vector to the interrupt at the end of the next instruction. It uses the value provided by `irq_num` to create a branch address into the Interrupt Vector Table (IVT). This branch address is simply 0x40 + ( `irq_num` * 4).

### 6.2.5 Non-Maskable Interrupt (NMI)

In the example IVC, `nmi` is connected to ivc_status[8]. Since this is derived from interrupt[8] which is not used by the example IVC, `nmi` is always inactive. If the NMI function is required by the ASIC designer, it should be connected to the desired signals (ensure that the sources are properly synchronised to clk).

## 6.3 Interrupt Timing

The xap4_single module samples `nmi`, `irq` and `irq_num` on the posedge of clk. It expects `irq` and `irq_num` to remain valid until it has branched to the IVT and interrupt software has written to the IVC to clear the respective bit in the ivc_status register. It is not necessary that `irq_num` remain stable throughout this period – the IVC may change `irq_num` to reflect the arrival of another interrupt with higher priority (an example of this is shown later in sections 6.3.4 and 6.3.5 ). However, any change on `irq_num` must settle in time for xap4_single to sample it reliably on each posedge of clk.

### 6.3.1 Interrupt latency

The XAP4 tests for exceptions and interrupts at the posedge of clk at the end of each instruction and at the end of each transfer in a blkcp (block copy) or blkst (block store) instruction. If the XAP4 detects `nmi` set and it is in User, Supervisor, or Interrupt Mode, it will take 5 clks to change to NMI Mode and save state to R0 and R6, after which the XAP4 will branch to the NMI entry in the IVT. Similarly, if the XAP4 detects `irq` set, interrupts are enabled in the FLAGS register, and the XAP4 is in User, Supervisor, or Interrupt Mode, it will take 5 clks to change to Interrupt Mode and save state to R0 and R6, after which the XAP4 will branch to the appropriate entry in the IVT.

The minimum interrupt latency is therefore 5 clk cycles, assuming that `irq` or `nmi` are asserted just as an instruction is completing, and that the system is using zero wait state memory. The maximum interrupt response time (assuming that the interrupt enable bit in the FLAGS register is set and that the XAP4 is not in NMI Mode) is 5+N+M, where N is the number of memory wait states and M is the length of the longest atomic instruction.

### 6.3.2 Single interrupt event

The following diagram show the timing for a single interrupt arriving at the IVC on `interrupt[4]`. This sets ivc_status[4] high at clock edge t1, which immediately sets `irq` high (this happens immediately because ivc_enable[4] is already set high). At the same time, the IVC sets `irq_num` to the priority code for interrupt[4]. Some time later, xap4_single recognises the interrupt, disables further interrupts and branches to the IVT. For `irq_num` = 0x6 for example, the branch address would be 0x40 + (0x6 * 4) = 0x58. Some time later at clock edge t2, the interrupt handler software writes to the ivc_clear register and sets bit 4. This clears ivc_status[4] which removes `irq`. Once `irq` is removed, the value on `irq_num` is ignored by xap4_single. Finally, the interrupt handler software executes an irqe instruction which re-enables interrupts.

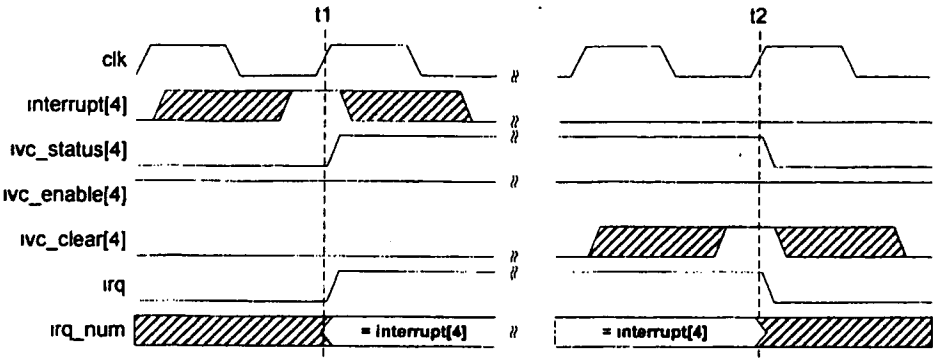



**Figure 6-1: Single interrupt**

### 6.3.3 Single interrupt, delayed enable

The following diagram show the timing for a single interrupt that is not initially enabled. As in the previous diagram, `interrupt[4]` arriving at the IVC sets ivc_status[4] high at clock edge t1. However, this does not affect `irq` and `irq_num`. yet, because ivc_enable[4] is low. Some time later at clock edge t2, the XAP4 software sets ivc_enable[4] high, which now allows the IVC to set `irq` and `irq_num`. Some time later, xap4_single recognises the interrupt, disables further interrupts and branches to the IVT. Some time later at clock edge t3, the interrupt handler software sets ivc_clear[4] high which clears ivc_status[4], removing `irq`. Finally, the interrupt handler software executes an irqe instruction which re-enables interrupts.

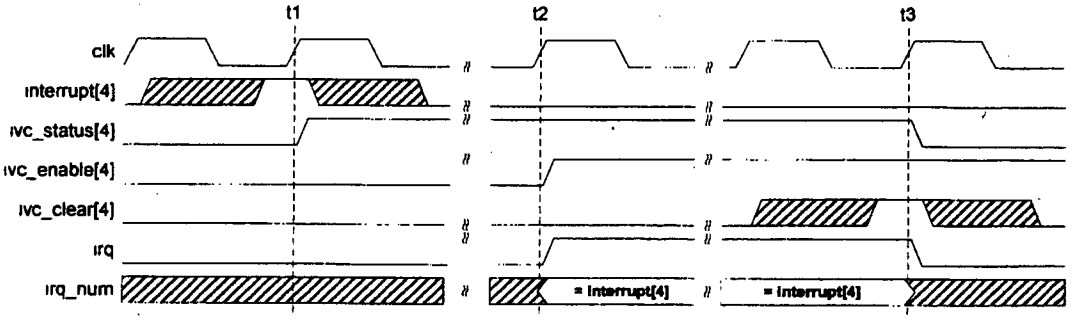



**Figure 6-2: Single interrupt, delayed enable**

### 6.3.4 Multiple interrupts, case 1

The following diagram show the timing for multiple interrupts. In this first case, a higher priority interrupt arrives after the XAP4 recognises the first interrupt, but before the XAP4 has cleared the first interrupt. For this example, the first (lower priority) interrupt is `interrupt[4]` and the second (higher priority) interrupt is `interrupt[7]`, and we will assume that both are already enabled in the IVC.

At clock edge t1, `interrupt[4]` arrives at the IVC and sets ivc_status[4] high. Some time later, xap4_single recognises the interrupt, disables further interrupts and branches to the IVT for interrupt[4]. Some time later at clock edge t2,

interrupt[7] arrives at the IVC and sets ivc_status[7] high, which changes irq_num to reference this higher priority interrupt. Some time later at clock edge t3, the interrupt handler software sets ivc_clear[4] high which clears ivc_status[4]. However, this does not remove irq, which is still set due to interrupt[7]. When the interrupt handler software executes an irqe instruction to re-enable interrupts, it will now recognise this second interrupt and branch to its interrupt handler code.

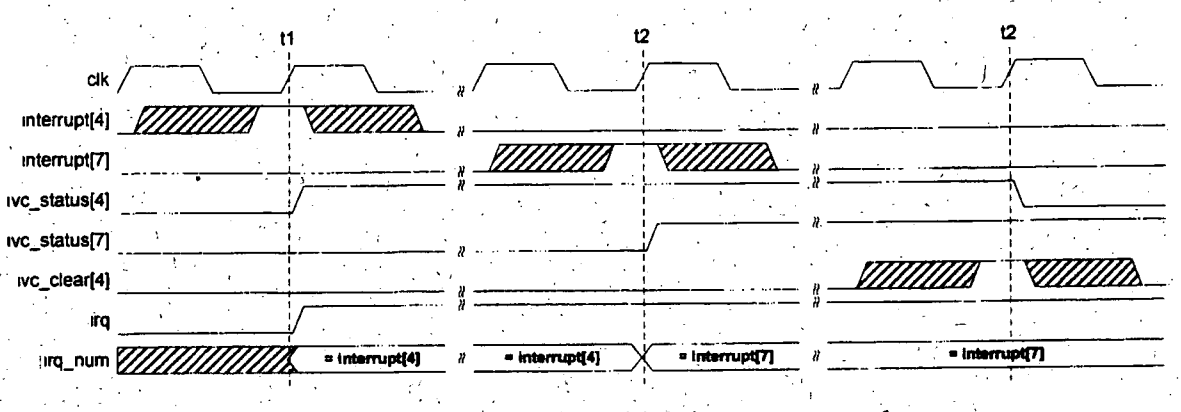



**Figure 6-3: Multiple interrupts, case 1**

### 6.3.5 Multiple interrupts, case 2

In this second case, a higher priority interrupt arrives before the XAP4 recognises the first interrupt and therefore gets serviced first. As for case 1, the first (lower priority) interrupt is interrupt[4] and the second (higher priority) interrupt is interrupt[7], and we will assume that both are already enabled in the IVC.

At clock edge t1, interrupt[4] arrives at the IVC and sets ivc_status[4] high. Some time later at clock edge t2, interrupt[7] arrives at the IVC and sets ivc_status[7] high, which changes irq_num to reference this higher priority interrupt. Some time later, xap4_single recognises the interrupt, disables further interrupts and branches to the IVT for interrupt[7]. Some time later at clock edge t3, the interrupt handler software sets ivc_clear[7] high which clears ivc_status[7]. However, this does not remove irq, which is still set due to interrupt[4]. When the interrupt handler software executes an irqe instruction to re-enable interrupts, it will now recognise the original first interrupt and branch to its interrupt handler code.

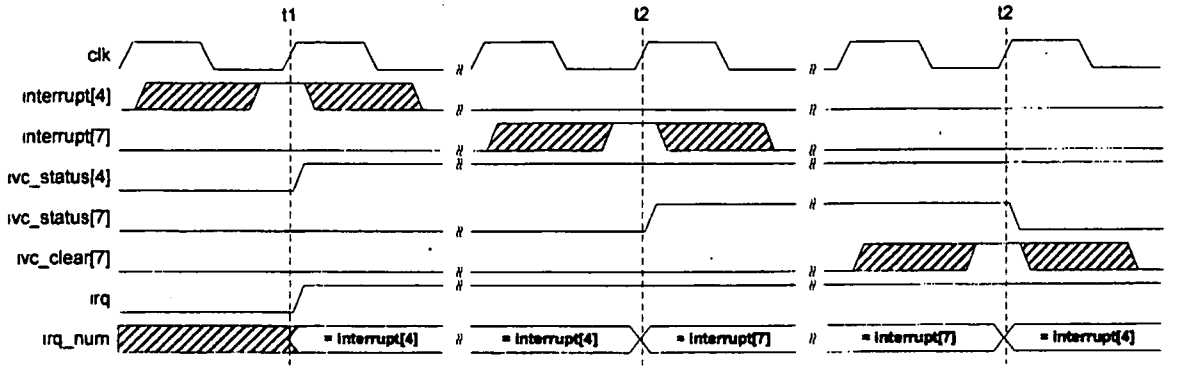



**Figure 6-4: Multiple interrupts, case 2**

### 6.3.6 NMI timing

The following diagram show the timing for an NMI. In the example IVC, `nmi` is connected to ivc_status[8], so it set by `interrupt[8]` and cleared by ivc_clear[8]. In a customer design, `nmi` can be connected to any appropriate signal source (properly synchronised to clk).

When the IVC detects a pulse on `interrupt[8]` at clock edge t1, it sets ivc_status[8] high. This immediately sets `nmi` high (this happens immediately because NMI is by definition always enabled). At the end of the next XAP4 instruction, xap4_single recognises the NMI, disables further interrupts and branches to the IVT. The branch address for an NMI= 0x04. Some time later at clock edge t2, the nmi handler software writes to the ivc_clear register and sets bit 8. This clears ivc_status[8] which removes `nmi`.

In some implementations, an NMI may be considered a fatal error or a shutdown request, recoverable only through a reset. In this case, it would not be necessary for the IVC module need to support a write-to-clear mechanism for `nmi`. Instead, it could be left set high until cleared by a system reset.

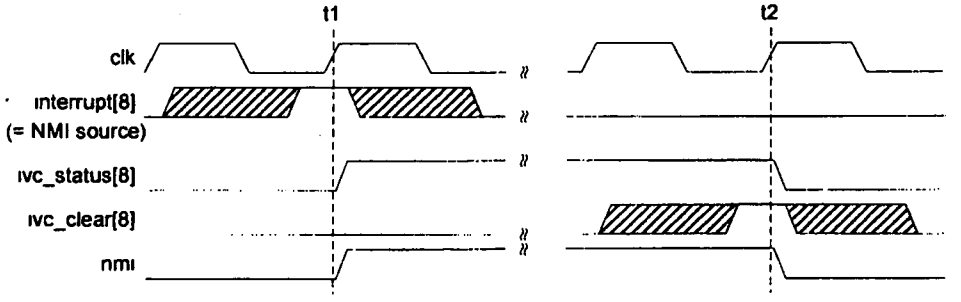



**Figure 6-5: NMI**

# 7 External Signal Descriptions

This section describes the external signals for the xap4_system module.

## 7.1 Time_stamp

| Signal Name | Direction | Description |
|---|---|---|
| `stat[8:0]` | input | General purpose status inputs; can be read in bits 8:0 of the status returned to xSIF or iSIF. This is typically used as a TimeStamp, driven by a free-running counter. |

## 7.2 Interrupts

| Signal Name | Direction | Description |
|---|---|---|
| `interrupt[15:0]` | input | External interrupt sources (up to 16). Interrupt events are detected and synchronised to `clk` before routing to the IVC. (In the example design, this synchronisation and edge detection is done in the clocks_and_resets module). If the IVC detects a logic high level on any interrupt at posedge clk, it captures this for use in generating `irq_num[3:0]` and `irq` to xap4_single. |

-1079-

## 7.3 xSIF interface

| Signal Name | Direction | Description |
|---|---|---|
| sif_cs | input | Chip select signal for SIF. Active high.<br>When SIF Pod has to support more than one SIF Slave ASIC, sif_cs must be brought off-chip. i.e. ASIC has 5 SIF pins.<br>When SIF Pod only has to support one SIF Slave ASIC, sif_cs should be tied high inside the ASIC. i.e. ASIC has 4 SIF pins. |
| sif_clk | input | Clock used by the SIF. The frequency of this clock is restricted to ≤ 1 MHz in command mode, but may be up to 50 MHz in normal mode. |
| sif_mosi | input | MasterOut, Slave In data signal: clocked into SIF logic on falling edge of sif_clk. |
| sif_loadb_in | input | This returns the state of the sif_loadb signal at the ASIC top-level. If this signal is set low and sif_loadb_out is not set low, then the SIF logic can detect that the SIF master is driving sif_loadb. |
| sif_loadb_out | output | This sets low the active-low handshake signal for SIF instructions (Normal mode SIF). It is the input to an open-drain driver for the bidirectional sif_loadb pin at the ASIC top-level. |
| sif_miso_enable | output | MasterIn, SlaveOut enable signal: when set high, this enables the tristate driver for sif_miso at the ASIC top-level. Is controlled by sif_cs. |
| sif_miso_out | output | MasterIn, SlaveOut data signal: clocked out of SIF logic on rising edge of sif_clk. This is the input to a tristate driver for sif_miso at the ASIC top-level. |

## 7.4 iSIF interface

| Signal Name | Direction | Description |
|---|---|---|
| `isif_address[15:0]` | input | For normal SIF Instructions , this provides the byte address at which to read or write XAP memory space. For Debug SIF Instructions, this field provides an OpCode for a particular operation. Tie this field to all zeros if iSIF is unused. |
| `isif_control[7:0]` | input | [7]: 0 = read,<br>1 = write<br>[6]: unused<br>[5:4]: 0 = byte access,<br>1 = word access,<br>2, 3 = illegal<br>[3]: 0 = Normal iSIF instruction,<br>1 = Debug iSIF Instruction<br>[2:0]: processor ID = 0 for single XAP<br>Tie this field to all zeros if iSIF is unused. |
| `isif_data_write[15:0]` | input | This provides the data to write XAP memory space or XAP registers. Some Debug SIF Instructions require this field to be set to a specific value. Tie this field to all zeros if iSIF is unused. |
| `isif_data_read[15:0]` | output | This returns the read data from XAP memory space or XAP registers. This should be clocked into a register at posedge of `clk` when `isif_done` is high. |
| `isif_status[11:0]` | output | This returns the status from an iSIF read or write instruction. This should be clocked into a register at posedge of `clk` when `isif_done` is high. The bits are interpreted as follows:<br>[11]: Status parity – can usually be ignored<br>[10]: Read data parity – can usually be ignored<br>[9]: 0 = no error,<br>1 = error<br>[8:4]: If bit 9 = 1, this contains a 5-bit error code. Otherwise, this contains `stat[8:4]`.<br>[3:0]: Contains `stat[3:0]`. |

| Signal Name | Direction | Description |
|---|---|---|
| isif_load | input | This indicates the start of an iSIF instruction request. If the xap4_system samples isif_load high on posedge of clk, then it samples isif_address, isif_data_write and isif_control on the same posedge clk. Tie this signal to zero if iSIF is unused. |
| isif_cancel | input | After an appropriate timeout, the iSIF master can abort an iSIF access that has not responded (with isif_done) by setting iSIF cancel. If the xap4_system samples isif_cancel high on posedge of clk, it aborts the access and then sets isif_done = 1, with an error code in isif_status to indicate abnormal completion. |
| isif_done | output | When the xap_system complete an iSIF access or if the access is cancelled (either by isif_cancel or by a cancel command from xsif), it sets isif_done =1 for one clk period. This should be used as a clock enable by the iSIF master to capture isif_status and isif_data_read on posedge clk. |

## 7.5 Clocks & Resets

| Signal Name | Direction | Description |
|---|---|---|
| clk | input | This is the main system clock used by xap4_single, xap4_mmu, xap4_ivc, and the synchronous memory devices connected to the MMU. All flip-flops in xap4_system that use clk are clocked on its posedge. |
| resetb_xap | input | This reset is used by xap4_single (except for the SIF logic), xap4_mmu, xap4_ivc. It is an active-low asynchronous reset. It resets the xap4 processor, but does not change the SIF mode (debug or normal) or the XAP4 run state (run continuous, run-to-break, single-step, or stop). |
| resetb_sif | input | This reset is used by the SIF logic within xap4_single. It is an active-low asynchronous reset. |

| Signal Name | Direction | Description |
|---|---|---|
| rst_usergates<br>rst_xap<br>rst_sif<br>rst_offchip | output | These synchronous reset outputs are generated by specific SIF commands. See section 3.3 for recommended usage. |
| unknown_instruction | output | This signal is generated when the processor encounters an un-implemented op-code in a privileged mode (all modes except User Mode). This is typically considered a fatal error; this signal can be used to generate a system reset. |
| wake_up | input | This signals to the XAP that it should wake up out of a sleep state. Active-high. |
| force_stop | input | This signals to the XAP that it should stop unconditionally at the end of the next instruction. Active-high. A typical use of this signal is to hold a RAM-based system halted at reset until an external means (such as iSIF) has loaded code into the RAM. |
| self_force_stop | output | This synchronous output resets to high (active). To stop the XAP when resetb_sif is pulled low, connect the self_force_stop output to the force_stop input. The reset can later be removed by clearing self_force_stop by writing to a specific address from the iSIF or xSIF interface |
| asleep | output | This synchronous output is set high if the XAP is executing a 'sleep' or 'sleepsif' instruction |
| debug | output | This synchronous output is set high if the XAP is in debug mode (which can be set by the iSIF or xSIF interface). |
| stopped | output | This synchronous output is set high if the XAP is stopped. |

## 7.6 Memory interface

| Signal Name | Direction | Description |
|---|---|---|
| ram_data_read[15:0]<br>rom_data_read[15:0]<br>ioreg_data_read[15:0] | input | 16-bit read data. These 3 data busses are driven by the RAM, ROM, and I/O register blocks. The MMU multiplexes these to provide a single read data bus to xap4_single |
| data_write[15:0] | output | 16-bit write data to memory and I/O registers. Also to MMU and IVC. |
| address[15:0] | output | 16-bit byte address to memory or external registers. Used internally by the MMU and IVC.<br>16-bit memory devices should ignore address[0] and use byte[1:0] instead.<br>8-bit memory devices should use address[0]. |
| byte[1:0] | output | Byte selects: byte[0] is set high during the address cycle for read or write accesses to the low byte (data bits 7:0), and byte[1] is set high for read or write accesses to the high byte (data bits 15:8). Full word accesses (data bits 15:0) cause both byte select signals to be set high. |
| ram_cs<br>rom_cs<br>ioreg_cs<br>ivc_cs | output | Chip-selects. One of these signals is set high during the address cycle to select RAM, ROM, I/O registers, IVC registers, or internal MMU registers. For example, the default MMU implements the following memory map:<br>ROM (rom_cs): not implemented<br>RAM (ram_cs): 0x0000 – 0xFEFF<br>I/O (ioreg_cs): 0xFF00 – 0xFF7F<br>IVC (ivc_cs): 0xFF80 – 0xFFBF<br>MMU regs : 0xFFC0 – 0xFFFF |
| we | output | Write-enable. This signal is set high during the address cycle for a write access. It is set low at all other times (during read access and no access). |

# 8 Internal Signal Descriptions

This section describes the internal signals within the xap4_system module.

## 8.1 Interface between xap4_single and xap4_ivc

| Pin Name | Direction | Description |
|---|---|---|
| irq | xap4_ivc to xap4_single | Interrupt request. Active-high. This signals to the core that there is a pending interrupt request. |
| irq_num[3:0] | xap4_ivc to xap4_single | Interrupt number. This is generated by the IVC from the incoming interrupt sources, based on the chosen priority scheme. It is used by the core to help form the address into the interrupt vector table. |
| nmi | xap4_ivc to xap4_single | Non-Maskable Interrupt. Active-high. |
| address[15:0] | xap4_single to xap4_ivc | See description in section 0 |
| data_write[15:0] | xap4_single to xap4_ivc | See description in section 0 |

## 8.2 Interface between xap4_mmu and xap4_ivc

| Pin Name | Direction | Description |
|---|---|---|
| ivc_data_read[15:0] | xap4_ivc to xap4_mmu | Data bus used to read registers in IVC. |
| byte[1:0] | xap4_mmu to xap4_ivc | See description in section 0 |
| we | xap4_mmu to xap4_ivc | See description in section 0 |
| ivc_cs | xap4_mmu to xap4_ivc | See description in section 0 |

## 8.3 Interface between xap4_single and xap4_mmu

| Pin Name | Direction | Description |
|---|---|---|
| prog_datab | xap4_single to xap4_mmu | This signal is high for a program (instruction) access, and low for a data access. It can be used to distinguish between exception conditions that need to be flagged by the MMU. |
| read | xap4_single to xap4_mmu | This signal is high for a read access. |
| write | xap4_single to xap4_mmu | This signal is high for a write access. |
| size | xap4_single to xap4_mmu | This signal is high for a 16-bit access, and low for an 8-bit access. |
| sif_xapb | xap4_single to xap4_mmu | This signal is high for a sif access, and low for a xap4 access. It can be used to distinguish between exception conditions that need to be flagged by the MMU. |
| core_mode[1:0] | xap4_single to xap4_mmu | This bus shows the xap4 processor mode. It can be used to distinguish between exception conditions that need to be flagged by the MMU. The encoding is:<br>00 : Supervisor Mode<br>01 : User Mode<br>10 : Interrupt Mode<br>11 : NMI Mode |
| address[15:0] | xap4_single to | See description in section 0 |

| Pin Name | Direction | Description |
|---|---|---|
| | xap4_mmu | |
| data_write[15:0] | xap4_single to xap4_mmu | See description in section 0 |
| data_read[15:0] | xap4_mmu to xap4_single | This is the data read by the MMU from a previous read cycle. It is only valid if mem_wait is low. |
| mem_wait | xap4_mmu to xap4_single | This signal is set high by the MMU if a memory device or I/O registers need more then one cycle to provide or accept data. This signal may be tied low if all addressed devices accept write data or provide read data on the second clk edge after their chip select signal is set high. |
| data_exception | xap4_mmu to xap4_single | The 2 exception signals are only set high when the XAP4 is in user mode.<br>These 3 signals are set high by the MMU if there is a fatal error during a memory data or program access. They cause the XAP4a to start exception processing.<br>In the example MMU, prog_exception is set high if a program access is made to I/O register space; data_exception is set high if a data write access is made to ROM space. |
| prog_exception | | |
| sif_mmu_error | | |

# 9 Debug Support

## 9.1 Breakpoints

### 9.1.1 Software Breakpoints

Software breakpoints can be inserted by swapping normal instruction with the 'brk' instruction in memory. The 'xIDE for XAP4' debugger uses this method whenever possible (i.e. if programs are stored in RAM), as there is no limit as to how many breakpoints can be used. However, this method cannot be used with memory that does not support individual word writes, nor does it support conditional breaks.

### 9.1.2 Hardware Breakpoints

XAP4 also provides a hardware breakpoint capability. This can be used with any kind of memory, because the memory itself is not modified. It can also support real-time conditional breaks. However, XAP4 only supports a single hardware breakpoint address. The 'xIDE for XAP4' debugger only uses this method when it is unable to use software breakpoints.

The XAP4 contains one breakpoint register, BRK0, that holds the breakpoint address. This can be configured using the breakpoint-enable register, BRKE, to stop the processor or cause an exception when an address is read, written, or executed. The BRK0 and BRKE registers can be accessed by privileged modes using the `movr2b`, `movb2r`, `movs2r`, and `movr2s` instructions.

The bits in the BRKE register are described below.

| Bit | Name | Description |
|---|---|---|
| 0 | W0 | Break-on-write: if this bit is set, an instruction making a data write to the address matching the BRK0 register may cause an exception or halt the processor after the instruction has executed. The PC will point at the next instruction to be executed. |
| 1 | R0 | Break-on-read: if this bit is set, an instruction making a data read to the address matching the BRK0 register may cause an exception or halt the processor after the instruction has executed. The PC will point at the next instruction to be executed. |
| 2 | E0 | Break-on-execute: if this bit is set, an instruction fetch from the address matching the BRK0 register may cause an exception or halt the processor before the instruction is executed. (This also works if a breakpoint is set for the second word of a 32-bit instruction.) The PC will remain at the address of the instruction that caused the break. |

### 9.1.3 Breakpoint Event Handling

When a software or hardware breakpoint is detected by the XAP4, it can respond by taking an exception or by halting or by doing neither. The XAP4 will generate a break exception only if the XAP4 is in User Mode and the B flag in the FLAGS register is set. If these conditions are not met, then the XAP4 will halt if the Run State (set by the SIF interface) is Run-To-Break. If this condition is also not met, then the breakpoint will be ignored.

## 9.2 Single Step

The XAP4 supports software-controlled single-stepping. If the T bit in the FLAGS register is set, a single-step exception will be triggered after every User Mode instruction. If there is another exception pending, this will take precedence over the single-step exception. (This ensures that an exception caused by the instruction itself, such as a DivideByZero exception, is not lost when stepping.) The Link Register (R6) will point to the next instruction to be executed and R0 will point to the instruction that caused the exception. This capability supports remote debugging and single stepping of User Mode applications.

In addition, XAP4 supports hardware-based single-stepping when used with a SIF-based debugger such as 'xIDE for XAP4'. When the Run State is set by the SIF interface to Single-Step, the XAP4 will execute one instruction and then stop. Similarly to software-controlled single-step, the XAP4 will not stop immediately if there is an interrupt or exception pending; instead it will process the interrupt or exception to the point where it is about to branch into the IVT. The next step will therefore execute the appropriate instruction in the IVT.

Block copy and block store instructions (blkcp and blkst) are non-atomic instructions, meaning that they can take an interrupt or exception after each transfer of the copy or store operation. This includes the ability to step through a block copy or store instruction one transfer at a time.

Single-stepping on a brk instruction or a hardware execution breakpoint will cause the PC to remain unchanged, until the break condition is removed or disabled.

## 9.3 The sif instruction

The XAP4 provides a sif instruction that can be used to provide predictable time-slots for non-invasive debugger monitoring. In a typical application, the user would insert one or more sif instructions (contiguous or dispersed) in the code to provide the desired debug bandwidth. When the XAP4 encounters a sif instruction and is not in User Mode, it sends a signal to the SIF interface that a fixed-length time-slot is available for a single SIF access. If there is a SIF access waiting, it will use this time-slot to read or write memory space, or read a XAP4 register. At the end of the time-slot, the XAP4 continues with the next instruction.

The length of the sif instruction is configured according to the maximum number of wait states that may be encountered by the sif access. This ensures that if a sif access does occur, it will have completed by the end of its time-slot. Consequently, the XAP4 program timing is unaffected by whether or not a sif access actually occurs during a sif instruction.

The sif instruction length can be tailored to the number of memory wait states by including a file called **xap4_user_constants.h**, This should include a statement to indicate the maximum number of memory wait states:

`define MAX_WAIT_STATES 5'bnnnnn

where nnnnn is a binary number in the range 00000 to 11110 (0 to 30 decimal). The number of clock cycles taken by the sif instruction is (MAX_WAIT_STATES * 2) + 4. If the **xap4_user_constants.h** file is not present, then the XAP4 assumes that the default sif instruction length need only support zero wait-state memory.

# 10 Simulation

Test benches have been run in Synopsys VCS. This includes a full instruction test.

Details are provided in the XAP4a Hardware QuickStart Guide (C7432-UM-001).

# 11 Synthesis

The whole of xap4_system has been implemented in synthesisable Verilog RTL.

## 11.1 ASIC

ASIC synthesis scripts are provided for Synplify-ASIC and Synopsys Design Compiler.

Details are provided in the XAP4a Hardware QuickStart Guide (C7432-UM-001).

## 11.2 FPGA

Xilinx Virtex2 synthesis script is provided for Synplify-Pro.

Xilinx Virtex2 Place & Route script is provided for Xilinx ISE.

Details are provided in the XAP4a Hardware QuickStart Guide (C7432-UM-001).

# 12 Directory Structure

The figure below shows the organisation of the deliverables in the XAP4a tar file.

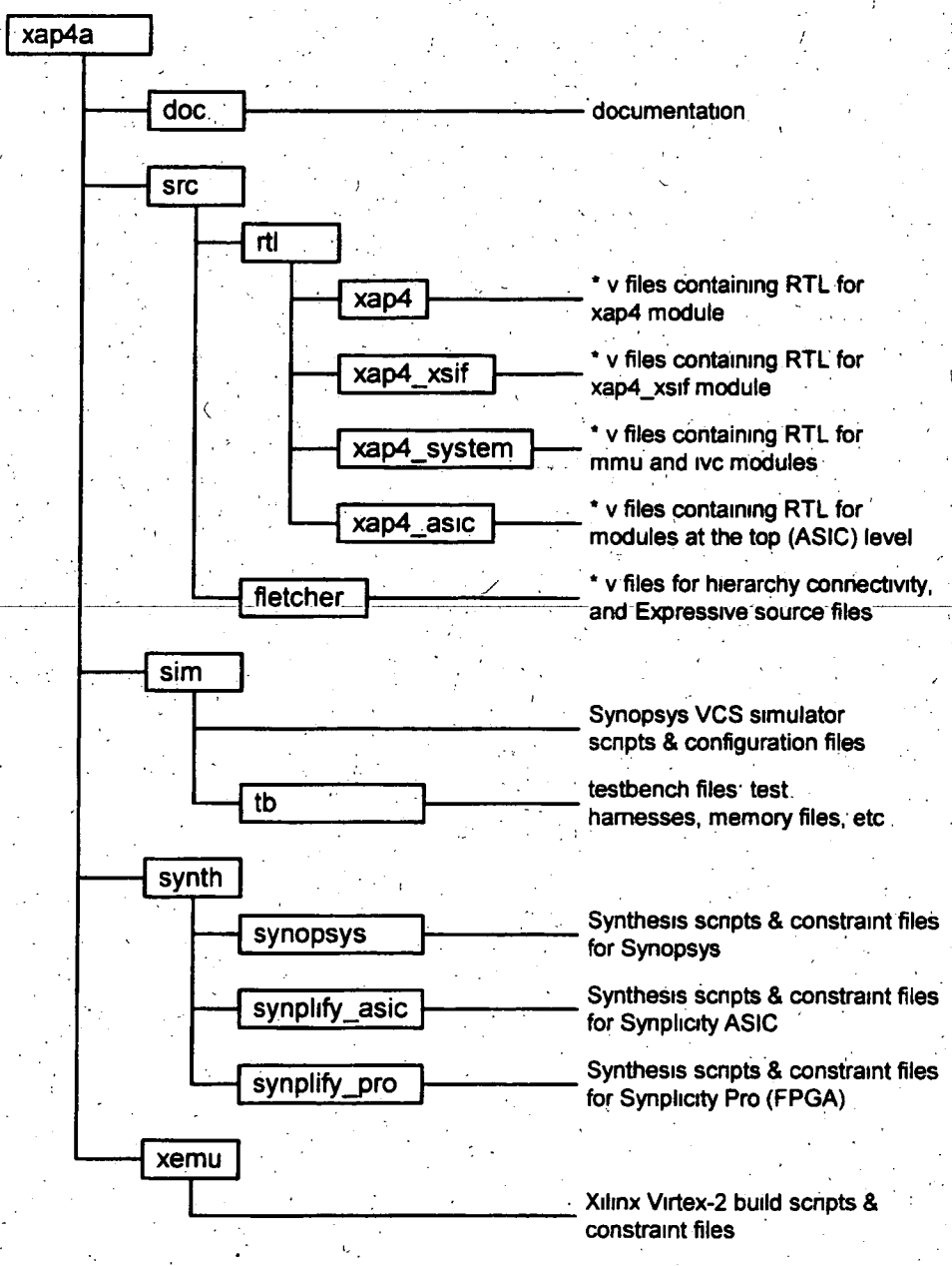



**Figure 12-1: Directory Structure**

Cambridge Consultants

# XAP4a Instruction Set Quick Reference Card

| Mnemonic | Operands | Operation | Flags | Cycles 16-bit | Cycles 32-bit |
|---|---|---|---|---|---|
| **Branches** | | | | | |
| **Unconditional** | | | | | |
| bra.a | #address | PC = #address | ---- | - | 3 |
| bra.p | #offset | PC = PC + #offset | ---- | 2 | 3 |
| bra.p.2 | #offset | PC = PC + #offset | ---- | 2 | 3 |
| bra.p.4 | #offset | PC = PC + #offset | ---- | 2 | 3 |
| bra.r | Rs | PC = Rs | ---- | 2 | - |
| bsr.a | #address | PC = #address | ---- | - | 4 |
| bsr.p | #offset | PC = PC + #offset | ---- | - | 4 |
| bsr.r | Rs | PC = Rs | ---- | 3 | - |
| **Conditional** | | | | | |
| bcc | #offset | If (C == 0); PC = PC + #offset | ---- | - | 2 / 3 |
| bcs | #offset | If (C == 1); PC = PC + #offset | ---- | - | 2 / 3 |
| beq | #offset | If (Z == 1); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| bge.s | #offset | If (N == V); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| bge.u | #offset | If (C == 0); PC = PC + #offset | ---- | - | 2 / 3 |
| bgt.s | #offset | If (N == V) && (Z == 0); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| bgt.u | #offset | If (C == 0) && (Z == 0); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| ble.s | #offset | If (N != V) \|\| (Z == 1); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| ble.u | #offset | If (C == 1) \|\| (Z == 1); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| blt.s | #offset | If (N != V); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| blt.u | #offset | If (C == 1); PC = PC + #offset | ---- | - | 2 / 3 |
| bmi | #offset | If (N ==1); PC = PC + #offset | ---- | - | 2 / 3 |
| bne | #offset | If (Z == 0); PC = PC + #offset | ---- | 1 / 2 | 2 / 3 |
| bpl | #offset | If (N == 0); PC = PC + #offset | ---- | - | 2 / 3 |
| bvc | #offset | If (V == 0); PC = PC + #offset | ---- | - | 2 / 3 |
| bvs | #offset | If (V == 1); PC = PC + #offset | ---- | - | 2 / 3 |

**Note:** The number of cycles = a / b. a = branch not taken. b = branch taken.

| Load and Store | | | | | |
|---|---|---|---|---|---|
| ld.8z.i | Rd, @(offset, Ra) | Rd = (uint16) (* (int8*) ( offset + Ra ) ) | ZN-- | 2 | 3 |
| ld.i | Rd, @(offset, Ra) | Rd = * (int16*) ( offset + Ra ) | ZN-- | 2 | 3 |
| ld.8z.r | Rd, @(Rx, Ra) | Rd = (uint16) (* (int8*) (Rx + Ra) | ZN-- | 2 | - |
| ld.r | Rd, @(Rx, Ra) | Rd = * (int16*) ( 2*Rx + Ra ) | ZN-- | 2 | - |
| ld.8z.p | Rd, @(offset) | Rd = (uint16) (* (int8*) ( offset + PC ) ) | ZN-- | - | 3 |
| ld.p | Rd, @(offset) | Rd = * (int16*) ( offset + PC ) | ZN-- | - | 3 |
| st.8.i | Rs, @(offset, Ra) | * (int8*) ( offset + Ra ) = Rs[7:0] | ---- | 2 | 3 |
| st.i | Rs, @(offset, Ra) | * (int16*) ( offset + Ra ) = Rs | ---- | 2 | 3 |
| stz.8.i | Rs, @(offset, Ra) | * (int8*) ( offset + Ra ) = 0x00 | ---- | - | 3 |
| stz.i | Rs, @(offset, Ra) | * (int16*) ( offset + Ra ) = 0x0000 | ---- | 2 | 3 |
| st.8.r | Rs, @(Rx, Ra) | * (int8*) ( Rx + Ra ) = Rs[7:0] | ---- | 2 | - |
| st.r | Rs, @(Rx, Ra) | * (int16*) ( 2*Rx + Ra ) = Rs | ---- | 2 | - |
| stz.8.r | Rs, @(Rx, Ra) | * (int8*) ( Rx + Ra ) = 0x00 | ---- | - | 3 |
| stz.r | Rs, @(Rx, Ra) | * (int16*) ( 2*Rx + Ra ) = 0x0000 | ---- | - | 3 |
| st.8.p | Rs, @(offset) | * (int8*) ( offset + PC ) = Rs[7:0] | ---- | - | 3 |
| st.p | Rs, @(offset) | * (int8*) ( offset + PC ) = Rs[7:0] | ---- | - | 3 |
| stz.8.p | Rs, @(offset) | * (int8*) ( offset + PC ) = 0x00 | ---- | - | 3 |


© Cambridge Consultants Ltd, 2005

Cambridge Consultants

# XAP4a Instruction Set Quick Reference Card

| Mnemonic | Operands | Operation | Flags | Cycles 16-bit | Cycles 32-bit |
|---|---|---|---|---|---|
| stz.p | Rs, @(offset) | * (int16*) ( offset + PC ) = 0x0000 | ---- | - | 3 |
| swap.i | Rd, @(0, Ra) | Swap Register with memory: *Ra <-> Rd | ZN-- | - | 5 |

**Note:** The figures for number of cycles assume aligned memory accesses. Add one cycle per unaligned access.

## Push and Pop

| | | | | | |
|---|---|---|---|---|---|
| push | RegList, #offset | push to stack | ---- | 2 - 8 | - |
| pop | RegList, #offset | pop from stack | ---- | 2 - 9 | - |
| pop.ret | RegList, #offset | pop from stack and return | ---- | 4 - 11 | - |

**Note:** The figures for number of cycles assume aligned memory accesses. Add one cycle per unaligned access.

## Move

| | | | | | |
|---|---|---|---|---|---|
| mov.i | Rd, #imm | Rd = #imm | ZN-- | 1 | 2 |
| mov.p | Rd, #offset | Rd = Rd + PC + #offset | ---- | - | 2 |
| mov.r | Rd, Rs | Rd = Rs | ZN-- | 1 | - |
| mov.32.r | Rd, Rs | {R(d+1), Rd} = {R(s+1), Rs} | ZN-- | 1 | - |

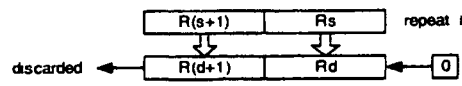
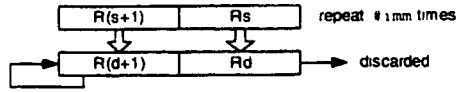
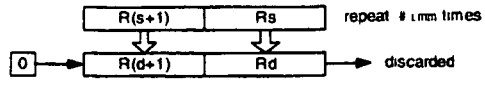
## ALU Operations

| | | | | | |
|---|---|---|---|---|---|
| **Add** | | | | | |
| add.c.i | Rd, Rs, #imm | Rd = Rs + #imm + C | ZNCV | - | 2 |
| add.c.r | Rd, Rs, Rt | Rd = Rs + Rt + C | ZNCV | 1 | - |
| add.i | Rd, Rs, #imm | Rd = Rs + #imm | ZNCV | 1 | 2 |
| add.r | Rd, Rs, Rt | Rd = Rs + Rt | ZNCV | 1 | - |
| **Subtract** | | | | | |
| sub.c.r | Rd, Rs, #imm | Rd = Rs - #imm - C | ZNCV | 1 | - |
| sub.r | Rd, Rs, Rt | Rd = Rs - Rt - C | ZNCV | 1 | - |
| sub.x.i | Rd, Rs, #imm | Rd = Rs - #imm | ZNCV | - | 2 |
| sub.xc.i | Rd, Rs, Rt | Rd = Rs - Rt | ZNCV | - | 2 |
| **Logical** | | | | | |
| and.i | Rd, Rs, #imm | Rd = Rs & Rt | ZN-- | 1 | 2 |
| and.r | Rd, Rs, Tr | Rd = Rs & Rt | ZN-- | 1 | - |
| or.i | Rd, Rs, #imm | Rd = Rs \| #imm | ZN-- | 1 | 2 |
| or.r | Rd, Rs, Rt | Rd = Rs \| Rt | ZN-- | 1 | - |
| xor.i | Rd, Rs, #imm | Rd = Rs ^ #imm | ZN-- | - | 2 |
| xor.r | Rd, Rs, Rt | Rd = Rs ^ Rt | ZN-- | 1 | - |
| **Multiply** | | | | | |
| mult.32s.i | Rd, Rs, #imm | {R(d+1), Rd} = Rs * #imm | ZN-- | - | 17 |
| mult.32s.r | Rd, Rs, Rt | {R(d+1), Rd} = Rs * Rt | ZN-- | - | 17 |
| mult.32u.i | Rd, Rs, #imm | {R(d+1), Rd} = Rs * #imm | ZN-- | - | 17 |
| mult.32u.r | Rd, Rs, Rt | {R(d+1), Rd} = Rs * Rt | ZN-- | - | 17 |
| mult.i | Rd, Rs, #imm | Rd = bttm 16 bits of (Rs * #imm) | ZN-- | - | 17 |
| mult.sh.r | Rd, Rs, Rt, #imm | Rd = (Rs * Rt) >> #imm | ZN-V | - | 18 |
| mult.r | Rd, Rs, Rt | Rd = bttm 16 bits of (Rs * Rt) | ZN-- | 16 | - |
| **Divide and Remainder** | | | | | |
| div.32s.i | Rd, Rs, #imm | Rd = {R(s+1), Rs} / #imm | ZN-V | - | 20 |
| div.32s.r | Rd, Rs, Rt | Rd = {R(s+1), Rs} / Rt | ZN-V | - | 20 |
| div.32u.i | Rd, Rs, #imm | Rd = {R(s+1), Rs} / #imm | ZNC- | - | 18 |
| div.32u.r | Rd, Rs, Rt | Rd = {R(s+1), Rs} / Rt | ZNC- | - | 18 |
| div.s.i | Rd, Rs, #imm | Rd = Rs / #imm | ZN-V | - | 20 |


© Cambridge Consultants Ltd, 2005

Cambridge Consultants

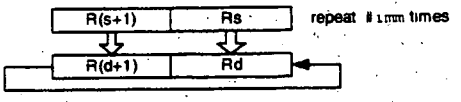
# XAP4a Instruction Set Quick Reference Card

| Mnemonic | Operands | Operation | Flags | Cycles 16-bit | Cycles 32-bit |
|---|---|---|---|---|---|
| div.s.r | Rd, Rs, Rt | Rd = Rs / Rt | ZN-V | - | 20 |
| div.u.i | Rd, Rs, #imm | Rd = Rs / #imm | ZNC- | - | 18 |
| div.u.r | Rd, Rs, Rt | Rd = Rs / Rt | ZNC- | - | 18 |
| divrem.32s.i | Rd, Rs, #imm | Rd = {R(s+1), Rs} / #imm ; R(d+1) = {R(s+1), Rs} % #imm | ---V | - | 20 |
| divrem.32s.r | Rd, Rs, Rt | Rd = {R(s+1), Rs} / Rt ; R(d+1) = {R(s+1), Rs} % Rt | ---V | - | 20 |
| divrem.32u.i | Rd, Rs, #imm | Rd = {R(s+1), Rs} / #imm ; R(d+1) = {R(s+1), Rs} % #imm | --C- | - | 18 |
| divrem.32u.r | Rd, Rs, Rt | Rd = {R(s+1), Rs} / Rt ; R(d+1) = {R(s+1), Rs} % Rt | --C- | - | 18 |
| divrem.s.i | Rd, Rs, #imm | Rd = Rs / #imm ; R(d+1) = Rs % #imm | ---V | - | 20 |
| divrem.s.r | Rd, Rs, Rt | Rd = Rs / Rt ; R(d+1) = Rs % Rt | ---V | - | 20 |
| divrem.u.i | Rd, Rs #imm | Rd = Rs / #imm ; R(d+1) = Rs % #imm | --C- | - | 18 |
| divrem.u.r | Rd, Rs, Rt | Rd = Rs / Rt ; R(d+1) = Rs % Rt | --C- | - | 18 |
| rem.32s.i | Rd, Rs, #imm | Rd = {R(s+1), Rs} % #imm | ZN-V | - | 20 |
| rem.32s.r | Rd, Rs, Rt | Rd = {R(s+1), Rs} % Rt | ZN-V | - | 20 |
| rem.32u.i | Rd, Rs, #imm | Rd = {R(s+1), Rs} % #imm | ZNC- | - | 18 |
| rem.32u.r | Rd, Rs, Rt | Rd = {R(s+1), Rs} % Rt | ZNC- | - | 18 |
| rem.s.i | Rd, Rs, #imm | Rd = Rs % #imm | ZN-V | - | 20 |
| rem.s.r | Rd, Rs, Rt | Rd = Rs % Rt | ZN-V | - | 20 |
| rem.u.i | Rd, Rs, #imm | Rd = Rs % #imm | ZNC- | - | 18 |
| rem.u.r | Rd, Rs, Rt | Rd = Rs % Rt | ZNC- | - | 18 |
| **Compare Operations** | | **Flags ==** | | | |
| cmp.8.i | Rs, imm[7:0] | Rs - #imm | ZNCV | 1 | 2 |
| cmp.8.r | Rs, Rt | Rs – Rt | ZNCV | 1 | - |
| cmp.8c.i | Rs, #imm[7:0] | Rs - #imm – C | ZNCV | - | 2 |
| cmp.8c.r | Rs, Rt | Rs – Rt – C | ZNCV | 1 | - |
| cmp.8x.i | Rs, #imm[7:0] | #imm – Rs | ZNCV | - | 2 |
| cmp.8xc.i | Rs, #imm[7:0] | #imm – Rs – C | ZNCV | - | 2 |
| cmp.c.i | Rs, #imm[15:0] | Rs - #imm | ZNCV | - | 2 |
| cmp.c.r | Rs, Rt | Rs – Rt | ZNCV | 1 | - |
| cmp.i | Rs, #imm[15:0] | Rs - #imm – C | ZNCV | 1 | 2 |
| cmp.r | Rs, Rt | Rs – Rt – C | ZNCV | 1 | - |
| cmp.x.i | Rs, #imm[15:0] | #imm – Rs | ZNCV | - | 2 |
| cmp.xc.i | Rs, #imm[15:0] | #imm – Rs – C | ZNCV | - | 2 |
| **Shift and Rotate** | | | | | |
| shiftl.32.i | Rd, Rs, #imm[4:0] |   | ZNC- | - | 2 |
| shiftl.32.r | Rd, Rs, Rt | As above but repeated Rt[4:0] times. | ZNC- | - | 2 |
| shiftl.i | Rd, Rs, #imm[3:0] | As shiftl.32.i, but not including R(s+1) → R(d+1) | ZNC- | 1 | - |
| shiftl.r | Rd, Rs, Rt | As above but repeated Rt[3:0] times. | ZNC- | 1 | - |
| shiftr.32s.i | Rd, Rs, #imm[4:0] |   | ZNC- | - | 2 |
| shiftr.32s.r | Rd, Rs, Rt | As above but repeated Rt[4:0] times. | ZNC- | - | 2 |
| shiftr.32u.i | Rd, Rs, #imm[4:0] |   | ZNC- | - | 2 |


© Cambridge Consultants Ltd, 2005

Cambridge Consultants

# XAP4a Instruction Set Quick Reference Card

| Mnemonic | Operands | Operation | Flags | Cycles 16-bit | Cycles 32-bit |
|---|---|---|---|---|---|
| shiftr.32u.r | Rd, Rs, Rt | As above but repeated Rt[4:0] times. | ZNC- | - | 2 |
| shiftr.s.i | Rd, Rs, #imm[3:0] | As shiftr.32s.i, but not including R(s+1) → R(d+1) | ZNC- | 1 | - |
| shiftr.s.r | Rd, Rs, Rt | As above but repeated Rt[3:0] times. | ZNC- | 1 | - |
| shiftr.u.i | Rd, Rs, #imm[3:0] | As shiftr.32u.i, but not including R(s+1) → R(d+1) | ZNC- | 1 | - |
| shiftr.u.r | Rd, Rs, Rt | As above but repeated Rt[3:0] times. | ZNC- | 1 | - |
| rotatel.32.i | Rd, Rs, #imm[4:0] |   | ZNC- | - | 2 |
| rotatel.32.r | Rd, Rs, Rt | As above but repeated Rt[4:0] times. | ZNC- | - | 2 |
| rotatel.i | Rd, Rs, #imm[3:0] | As rotatel.32.i, but not including R(s+1) → R(d+1) | ZNC- | 1 | - |
| rotatel.r | Rd, Rs, Rt | As above but repeated Rt[3:0] times. | ZNC- | 1 | - |
| **Block Copy and Store** | | | | | |
| blkcp.r | Rd, Rs, Rt | **while**(Rt > 0) { Rt— *Rd++ = *Rs++ } | ---- | - | 2n + 8 |
| blkcps.r | Rd, Rs, Rt | **while**(Rt > 0) { Rt-- **if** ( (*Rd++ = *Rs++ ) == '\0' ) **break** } | ---- | - | 2n + 8 |
| blkst.8.r | Rd, Rs, Rt | **while**(Rt > 0) { Rt— *Rd++ = Rs } | ---- | - | n + 9 |
| blkst.r | Rd, Rs, Rt | **while**(Rt > 0) { Rt— *Rd++ = Rs } | ---- | - | n + 9 |
| **Note:** n is the number of memory transfers. See C7432-UM-002 for more details. | | | | | |
| **Miscellaneous instructions** | | | | | |
| sext.r | Rd | Sign Extend: Set Rd[15:8] = Rd[7] | ZN-- | 1 | - |
| **System Instructions** | | | | | |
| brk | - | Break | ---- | 1 | - |
| halt | - | Halt | ---- | 1 | - |
| nop | - | No Operation | ---- | 1 | - |
| print.r | Rs | Print Register | ---- | 1 | - |
| sleepnop | - | Sleep | ---- | 1 | - |
| sleepsif | - | Sleep and allow SIF | ---- | 1 | - |
| **Debug** | | | | | |
| c | Rd | Rd = XAP4 licence number | ---- | 1 | - |
| movb2r | Rd, BRs | Rd = BRs | ZN-- | - | 2 |
| movr2b | BRd, Rs | BRd = Rs | ---- | - | 2 |
| sif | - | Perform SIF cycle | ---- | 4 | 5 |
| ver | Rd | Rd = XAP4 version number | ---- | 1 | - |
| **Interrupt and exceptions** | | | | | |
| irqd | - | Disables Interrupts (FLAGS.E = 0) | ---- | 1 | - |
| irqe | - | Enables Interrupts (FLAGS.E = 1) | ---- | 1 | - |
| irqs | Rd | Interrupt Status (Rd = FLAGS.E ? 1 : 0) | ZN-- | 1 | - |
| rtie | - | Return from Interrupt/Exception | ZNCV | 2 | - |
| trap.i | #imm | Trap Immediate | ---- | - | 6 |
| trap.r | Rs | Trap Register | ---- | - | 6 |
| **Special Registers** | | | | | |
| movs2r | Rd, SRs | Rd = SRs | ZN-- | - | 2 |
| movr2s | SRd, Rs | SRd = Rs | ---- | - | 2 |

**Note:** All figures for number of cycles assume 16-bit single cycle memory is used.


**© Cambridge Consultants Ltd, 2005**

Cambridge Consultants

-1097-

## Technical Memorandum

**To**

**cc**

**Prepared by**

**Reviewed by**

**Subject: The SIF (Serial InterFace) Definition**

# Table of Contents


1 INTRODUCTION ..... 4
1.1 SIF HISTORY ..... 4
1.2 SIF APPLICATIONS ..... 5
*1.2.1 Software Debug* ..... 5
*1.2.2 Hardware Setup and Data Acquisition* ..... 5
*1.2.3 ASIC Production Test* ..... 5
*1.2.4 In-field Diagnostics* ..... 5
1.3 RELEASE HISTORY ..... 6
1.4 RELATED DOCUMENTS ..... 6
2 SIF OVERVIEW ..... 7
2.1 BLOCK DIAGRAM ..... 7
2.2 SIF MASTER READS/WRITES A SIF SLAVE ..... 7
2.3 SIF POD PROTOCOL ..... 8
2.4 16-IDC SIF INTERFACE ..... 8
2.5 API ..... 8
*2.5.1 xSIF API* ..... 8
*2.5.2 SIF API* ..... 9
2.6 SIF PODS ..... 9
*2.6.1 ISA SIF Pods (discontinued)* ..... 9
*2.6.2 LPT SIF Pod* ..... 10
*2.6.3 USB SIF Pod* ..... 10
*2.6.4 Universal SIF Pod* ..... 11
2.7 SIF TOOLKIT ..... 11
*2.7.1 Environment* ..... 11
*2.7.2 Drivers* ..... 11
*2.7.3 Application Tools* ..... 11
2.8 SOFTWARE APPLICATIONS THAT USE SIF ..... 12
*2.8.1 xIDE* ..... 12
*2.8.2 Matlab* ..... 12
3 16-IDC SIF INTERFACE ..... 13
3.1 PINOUT ..... 13
*3.1.1 SIF Slave Pad Types* ..... 17
3.2 CONNECTOR ..... 18
3.3 CABLE ..... 18
3.4 TERMINATION ..... 19
3.5 MAXIMUM SHIFT FREQUENCIES ..... 20
*3.5.1 Un-buffered Slave PCB (Production)* ..... 20
*3.5.2 Buffered Slave PCB (Test)* ..... 20

**4 SIF SLAVE PROTOCOL ..... 21**
4.1 PINS USED FOR NORMAL AND COMMAND MODES ..... 21
4.2 BIT ORDER ..... 21
4.3 SIF POD TIMING ..... 22
*4.3.1 SIF_CLK Duty Cycle ..... 22*
4.4 SIF READ ..... 22
*4.4.1 Single SIF Read ..... 23*
*4.4.2 Pipelined SIF Reads ..... 23*
4.5 SIF WRITE ..... 24
*4.5.1 Single SIF Write ..... 24*
*4.5.2 Pipelined SIF Writes ..... 24*
4.6 SYNCHRONISING A FIFO WITH SIF_LOADB FOR DATA ACQUISITION ..... 25
4.7 SIF CANCEL ..... 26
*4.7.1 SIF Cancel with SIF_CS ..... 27*
*4.7.2 SIF Cancel with SIF_CLK ..... 27*
4.8 SIF COMMAND ..... 27
**5 SIF ERROR PROCESSING ..... 28**
5.1 DATA INTEGRITY ..... 32
*5.1.1 Data Integrity between Computer and Pod ..... 32*
*5.1.2 Data Integrity between Pod and Slave ..... 32*
5.2 ERROR CORRECTION ..... 32
5.3 ERROR DETECTION ..... 32
*5.3.1 Parity Checking ..... 33*
*5.3.2 Data Integrity in Shift Registers and Cable ..... 33*
*5.3.3 Diagnostics ..... 33*
5.4 USER VIEW OF SIF ERROR-HANDLING ..... 34
**6 SIF NETWORKS ..... 34**
6.1 SINGLE MASTER, SINGLE SLAVE ..... 35
6.2 SINGLE MASTER, MULTI SLAVE ..... 35
6.3 MULTI MASTER, MULTI SLAVE ..... 36
6.4 SHIFT REGISTERS AND CLOCKS IN MASTER AND SLAVE ..... 37
6.5 SHIFT REGISTER TIMING DELAYS BETWEEN MASTER AND SLAVE ..... 38
6.6 MASTER-SLAVE HANDSHAKE ON SIF_LOADB ..... 39
6.7 PROTECTION, TERMINATION AND BUFFERING BETWEEN MASTER AND SLAVE ..... 40
**7 SIF SLAVE ..... 42**
7.1 BLOCK DIAGRAM ..... 44
7.2 SIF SLAVE CONTAINING MULTIPLE SIF PROCESSORS ..... 44
7.3 SIF PINS ..... 45
*7.3.1 SIF Slave Pin Order for Package and Silicon Die ..... 46*
*7.3.2 Sharing SIF Pins with JTAG ..... 46*
*7.3.3 Sharing SIF Pins with Scan Path ..... 46*
7.4 SHIFT REGISTER FIELDS ..... 47
*7.4.1 Address Field ..... 47*
*7.4.2 Control Field ..... 48*
*7.4.3 Data Field ..... 48*
*7.4.4 Status Field ..... 49*
7.5 XAP AND APE PROCESSORS ..... 49
7.6 VERILOG RTL IMPLEMENTATION ..... 51
**8 SIF SLAVE PCB ..... 51**
**..... 53**

8.1 STRATEGIES ..... 53
8.2 SINGLE SLAVE ..... 53
*8.2.1 Low-cost Production Board* ..... 54
*8.2.2 Test Board* ..... 54
8.3 MULTI SLAVE ..... 55
*8.3.1 Low-cost Production Board* ..... 56
*8.3.2 Test Board* ..... 56
**9 SIF MASTER ..... 57**
9.1 UNIQUE SIF POD IDENTIFIERS ..... 58
9.2 BLOCK DIAGRAM ..... 58
9.3 VERILOG RTL IMPLEMENTATION FOR FPGA ..... 58
*9.3.1 STOPB and RUN_STEP circuitry* ..... 59
9.4 ESD AND HIGH VOLTAGE PROTECTION IN SIF POD ..... 61
**10 ELECTRICAL ..... 62**
10.1 POWER SUPPLY ..... 63
10.2 INPUT VOLTAGE THRESHOLDS ..... 63
10.3 OUTPUT VOLTAGE LEVELS ..... 63
10.4 TIMING ..... 63
**11 ISIF – INTERNAL SIF ..... 63**
11.1 USAGE MODELS ..... 64
ISIF HARDWARE IMPLEMENTATION ..... 66
11.3 XSIF V ISIF PRIORITY ..... 67
11.4 PARITY BITS ..... 71
11.5 SIF COMMAND MODE ..... 71
11.6 SIF CANCEL ..... 71
11.7 XAP STATUS REGISTER ..... 72
11.8 CONNECTING AND USING ISIF ..... 72
..... 73

# 1 Introduction

SIF is a patented, non-invasive, real time, synchronous serial interface developed by Cambridge Consultants. SIF is used to debug and test ASICs, ASSP, SoC or FPGA.

The SIF Slave is typically an ASIC or FPGA. The SIF Master is typically a PC, Silicon Tester or Microprocessor. The SIF Master can read or write any memory-mapped address inside the SIF Slave in real time, without affecting the timing or functionality of the SIF Slave. This enables the SIF Master to setup and configure the SIF Slave and to do data-acquisition from it. The SIF Master can also develop and debug embedded software for the SIF Slave (debug functions include start, stop, run to breakpoint etc).

The standard SIF Slave interface has 4 pins; SIF_CLK, SIF_MOSI, SIF_MISO, SIF_LOADB. The first 3 pins are unidirectional signals to a shift register (which can be of different lengths for different circuits). SIF_LOADB is a bi-directional pin for hand-shaking. This interface will support a single-master, single-slave network. The addition of a 5th pin, SIF_CS to the SIF Slave enables single-master, multi-slave networks. The addition of the SIF_REQ, SIF_ACK pins to the SIF Master enables multi-master, multi-slave networks.

## 1.1 SIF History

| *Year* | *Name* | *Description* |
|---|---|---|
| 1993 | SIF first developed | Used in mixed analogue-digital industrial ASIC. |
| 1994 | SIF-ISA Card | ISA bus to 50-IDC SIF |
| 1994 | SIF-ISA Pod94 | 50-IDC SIF to 16-IDC SIF |
| 1994 | 16-IDC SIF Interface | Defined for use on SIF Slave and SIF Master circuit boards. Master and Slaves connected by 16-way ribbon cable (3 power, 4 SIF and 9 IO pins). |
| 1994 | XAP1 processor | Includes a 36-bit long SIF. xIDE uses SIF API. |
| 1995 | LPT SIF Pod | LPT to 16-IDC SIF |
| 1998 | SIF-ISA Pod98 | 50-IDC SIF to 16-IDC SIF |
| 1998 | SIF API + Toolkit | Windows. In C, MFC. |
| 1999 | XAP2 processor | Includes a 64-bit long SIF. xIDE uses SIF API. |
| 2003 | USB SIF Pod | USB2 (480 MHz) to 16-IDC SIF |
| 2004 | XAP3 processor | Includes an 88-bit long SIF. xIDE uses xSIF API. |
| 2005 | xSIF API + Toolkit | Windows, x86 Linux, MacOSX. In C++, Qt. |
| 2005 | Universal SIF Pod | Ethernet and USB to 16-IDC SIF |

| | | |
|---|---|---|
| 2005 | XAP4 processor | Includes a 52-bit long SIF. |

## 1.2 SIF Applications

The following sections describe typical ways in which the SIF is used, throughout the life of an ASIC or FPGA. The FPGA could be the final product or an emulator for an ASIC before tape-out.

### *1.2.1 Software Debug*

SIF Slave ASICs normally contain one or more microprocessors (e.g XAP1, XAP2, XAP3 or other). The SIF provides a debug interface for code development on these microprocessors and is used for functions such as :

- Code download
- Memory read and write
- Debug functions (start, stop, set breakpoint, run to breakpoint, ...)

### *1.2.2 Hardware Setup and Data Acquisition*

The hardware in an ASIC can often be configured by software to have alternative functional behaviours. The configuration is set by writing the required values to specific addresses. This can be done by the on-chip processor, or by an external SIF Master using the SIF interface.

It is often useful to transfer data generated in an ASIC to a computer for detailed analysis. For example it is useful to analyse an ADC output stream in Matlab. The SIF enables such data acquisition to be done at high speed (> 600k 32-bit words/s with USB SIF Pod).

### *1.2.3 ASIC Production Test*

SIF is often used by an ASIC tester (e.g Teradyne Catalyst) in production test. This is especially useful for the analogue sections of an ASIC. The SIF can be used to confgure the ASIC as required for a particular test, and then to read back the results. SIF can co-exist (and share 3 pins) with JTAG and ScanPath test interfaces.

### *1.2.4 In-field Diagnostics*

Support costs for a manufacturing company can be high when a product is out in the field (large volumes spread all over the world). If users have a problem with their product, the support costs are reduced if they can run software that provides good diagnostic information. Such computer software can access large regions of the ASIC (while it is running in normal functional mode) via the SIF. The acquired data can then be sent back to the product manufacturer to help them solve the problem.

## 1.3 Release History

| *Version* | *Date* | *Comment* |
|---|---|---|
| a | 21 Sep 1995 | C4862-TM-005. Original Release |
| 1.0 | 27 Mar 2002 | First DocMan Release. Same as C4862-TM-005a. |
| 2.0 | 22 April 2005 | Major release. Includes Command mode, SIF Cancel, Recommended hardware implementations for Master and Slave. |
| 3.0 | 25 Oct 2005 | Major Release. Adds iSIF. |

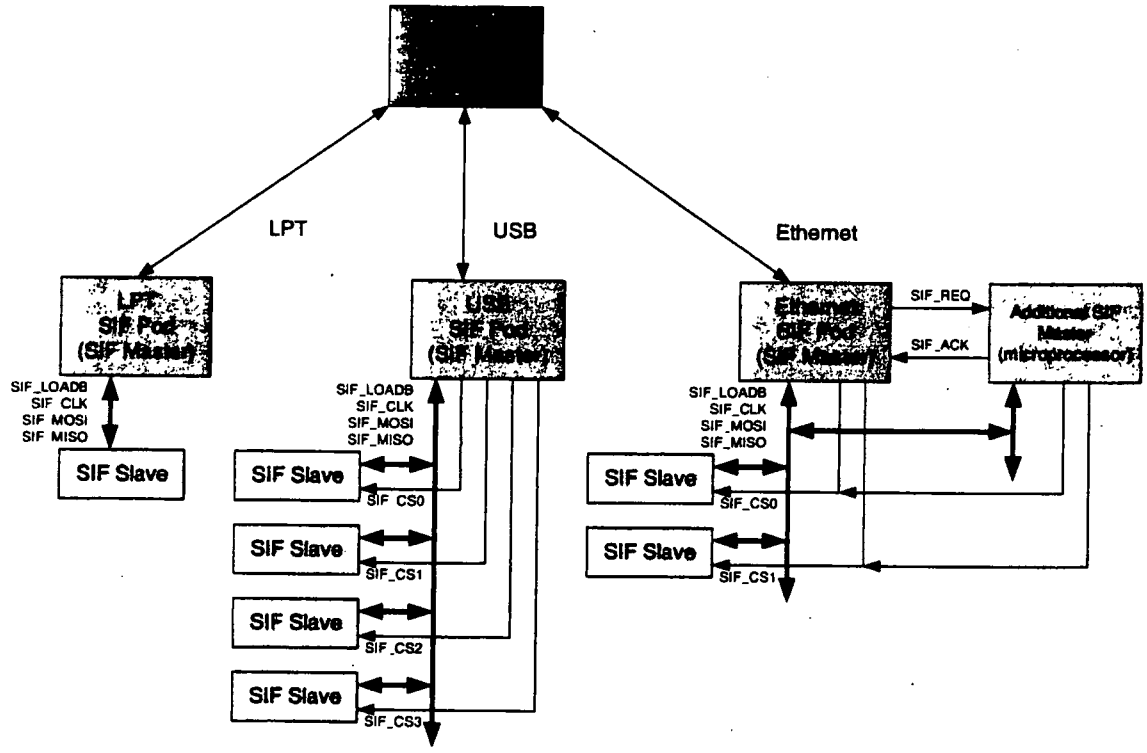
## 1.4 Related Documents

| *Reference* | *Title* | *Author* |
|---|---|---|
| C4862-TM-002 | PC-ISA SIF Pod98 | AGM |
| C4862-TM-003 | PC-LPT SIF Pod | AGM |
| C4862-TM-005 | SIF Definition (this document) | AGM |
| C4862-TM-009 | SIF Drivers (inc SIF API) | KLS |
| C4862-TM-011 | Standard ASIC/FPGA SIF Implementation | DML |
| C4862-TM-014 | SIF Timing and Signal Integrity | JPC2 |
| C4862-TM-016 | xSIF API | KLS |
| C4862-TM-017 | SIF Pod Protocol | KLS |

# 2 SIF Overview

## 2.1 Block Diagram

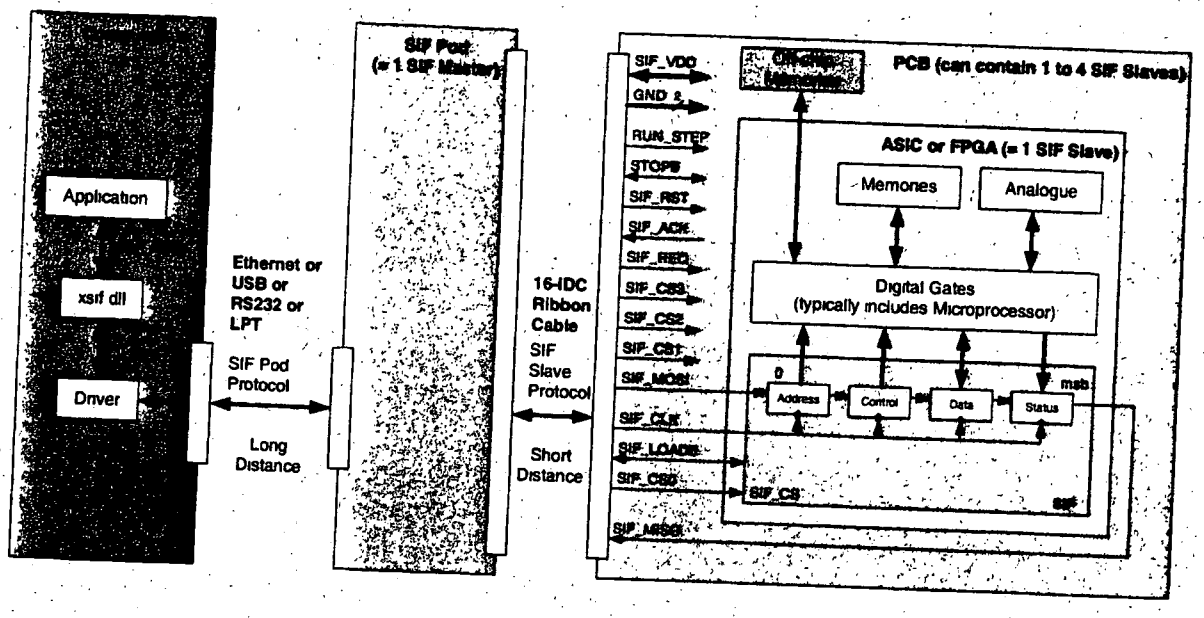
This diagram shows how a computer can read or write data to any of the SIF Slaves. The SIF Slaves can all be different devices running in their normal functional modes. The SIF reads and writes will not affect the timing or functionality of the SIF Slaves.




## 2.2 SIF Master reads/writes a SIF Slave

The purpose of the SIF is to let a SIF Master read or write data to one or more SIF Slaves. The reads and writes are always initiated by the SIF Master. The SIF Slave never initiates any reads or writes, it is only responsive. The data rates are determined by the SIF Master and can be faster or slower than the system clock used inside the SIF Slave.

The clever circuitry is in the SIF Slave. It is always implemented in hardware. It allows a SIF Master to read or write any address-mapped variable (normally a register or memory). The SIF Master is normally a computer which access the SIF Slave via a SIF Pod. The SIF Master functionality can be implemented in hardware or software.




## 2.3 SIF Pod Protocol

Old SIF Pods (ISA, LPT) do not have any intelligence. The pods are like combinational circuitry. There is no checking for data integrity between the computer and pod.

New SIF Pods (Universal = Ethernet, USB2, RS232) do have intelligence (contain a processor and FPGA). They support the new SIF Pod Protocol. This maintains data integrity between the computer and pod by sending packets with CRC codes.

The SIF Pod Protocol is defined in C4862-TM-017.

## 2.4 16-IDC SIF Interface

The 16-IDC SIF interface used between a SIF Pod and a PCB (or PCBs) containing 1 to 4 SIF Slaves. The interface contains 3 power, 4 SIF and 9 IO pins. This enables the pod/master to read and write up to 4 slaves in Normal or Command mode.

Pin 1 is SIF_VDD. This allows the Pod to either supply power to the Slave PCB or to sense the voltage being used on it.

## 2.5 API

Cambridge Consultants has developed computer drivers for the SIF. These support a number of operating systems and pods. The API (Application Programmer's Interface) for these drivers are :

### *2.5.1 xSIF API*

This is the new API for SIF applications :

- Defined in C4862-TM-016.

- Supports 4 SIF Shift Register fields (Address, Control, Data, Status).
- API for C, Matlab.
- 2005 to ?
- Used in SIF Toolkits 4.*.
- Implemented as xsif.dll on Windows PCs (Win2k, WinXP)
- Implemented as xsif.so on x86 Linux computers.
- Implemented as xsif.so on MacOSX computers.
- Used in 'xIDE for XAP3'.

### 2.5.2 SIF API

This is the old API for SIF applications :

- Defined in C4862-TM-009.
- Supports 2 SIF Shift Register fields (Address, Data).
- API for C, Basic, Matlab
- 1998 to 2004.
- Used in SIF Toolkits 1.*, 2.*, 3.*.
- Implemented as sif.dll on Windows PCs (Win98, WinNT, Win2k, WinXP).
- In SIF Toolkits 4.*, sif.dll is a thin 'twister' to xsif.dll.
- Used in 'xIDE for XAP1'.
- Used in 'xIDE for XAP2'.

## 2.6 SIF Pods

Cambridge Consultants manufactures a number of SIF hardware pods. These are used to connect computers (normally PCs) to SIF slaves. The connection from the pod to the PCB containing 1 to 4 SIF slaves is always a 16-IDC (ribbon and connectors). There are a variety of standard interfaces from the computer to the pod. SIF Toolkits contain drivers to support all these pods under a variety of computer operating systems.

### 2.6.1 ISA SIF Pods (discontinued)

There were 2 ISA SIF Pods :

- ISA SIF Pod 94
- ISA SIF Pod 98 (defined in C4862-TM-002)

Characteristics are :

- Both have been discontinued.
- Both connect to the ISA SIF Card (defined in C4862-TM-001) with a 50-IDC ribbon cable. The ISA SIF card is full length and plugs into a PC ISA Slot (not found in modern PCs). The signals in the 50-IDC interface are opto-isolated from the PC. The SIF_VDD and GND lines in the 50-IDC interface are DC-DC isolated from the PC.
- Can supply low power on pin 1 SIF_VDD (~ 100 mA). Or it can also sense the voltage at the Slave PCB. This option is set by jumpers.
- Serial shift rates up to 2M6 bits/s.
- Was used in many projects by Cambridge Consultants and clients (for development, functional test and production test).
- Supported in SIF Toolkit 3.*.
- Not supported in SIF Toolkit 4.*.

### 2.6.2 LPT SIF Pod

Defined in C4862-TM-003. This is the simplest SIF pod available :

- Pins 15 (STOPB) and 16 (RUN_STEP) in the 16-IDC are not connected. Therefore it cannot be used with XAP1.
- Can be thought of as combinational circuitry. Does not contain any state.
- Does not contain any microprocessor or FPGA.
- Can only supply very low power on pin 1 SIF_VDD (~ 10 mA). Or it can also sense the voltage at the Slave PCB. This option is set by a jumper.
- Serial shift rates up to 200k bits/s.

### 2.6.3 USB SIF Pod

Defined in C7028-S-001. This is the fastest SIF pod available :

- 480MHz USB2 bus-powered device.
- Enables 32-bit data acquisition at speeds up to 600kHz.
- Implements full 16-IDC SIF Interface.
- Can supply low power on pin 1 SIF_VDD (~ 100 mA). Or it can also sense the voltage at the Slave PCB. This option is set by configuration software on the computer.
- Serial shift rates up to 24M bits/s.

### 2.6.4 Universal SIF Pod

New pod for 2005 :

- Implements full 16-IDC SIF Interface.
- 10/100 BaseT Ethernet
- 480MHz USB2 (device and host, enabling daisy-chaining of pods)
- RS232
- Can take a DC power input.
- Can supply low power on pin 1 SIF_VDD (~ 100 mA). Or it can also sense the voltage at the Slave PCB. This option is set by configuration software on the computer.

## 2.7 SIF Toolkit

### 2.7.1 Environment

| *SIF Toolkit Version* | *Language* | *Graphics* | *API* | *SIF Shift Register Fields (length)* | *Supported Operating Systems* | *Supported Pods* |
|---|---|---|---|---|---|---|
| 3.* | C | MFC | SIF | Address (0 to 31)<br>WriteBit (1)<br>Data (0 to 32) | Win98<br>WinNT<br>Win2k<br>WinXP | ISA<br>LPT<br>USB |
| 4.* | C++ | Qt | xSIF | Address (0 to 32)<br>Control (0 to 32)<br>Data (0 to 32)<br>Status (0 to 32) | Win2k<br>WinXP<br>x86 Linux<br>MacOSX | LPT<br>USB<br>Ethernet<br>Universal |

### 2.7.2 Drivers

The new xSIF drivers run on Win2k, WinXP, x86 Linux PCs.

They implement the connection from xSIF API to the relevant communications socket (Ethernet, USB, RS232, LPT).

### 2.7.3 Application Tools

The major tools included in SIF Toolkit 3.* are :

- Configuration tool (enables manual setup for features that were not discovered automatically).
- Data Writer (one-shot reads and writes to specified slaves and addresses).
- Data Logger (continuous reads from specified slaves and addresses).

SIF Toolkit 4.* integrates the above functionality into a single tool called SIF Explorer. It also adds the following extra functionality :

- Discovery tool. Finds what SIF pods and Slaves are available on the network and auto-configures them where possible. Some old SIF Slaves may still need to be configured manually.

## 2.8 Software Applications that use SIF

### *2.8.1 xIDE*

xIDE is an Integrated Development Environment developed by Cambridge Consultants. It is implemented in C++ and Qt. It consists of a standard kernel and custom plugins for particular processors or complete ASICs.

xIDE is used as the IDE for the XAP1, XAP2 and XAP3 ASIC processors. They all include software simulators. They also include SIF interfaces via pods to hardware emulators (normally based on xEMU). These xIDE toolkits are major applications that use the SIF and xSIF APIs to the SIF drivers in Windows, x86 Linux and MacOSX computers.

### *2.8.2 Matlab*

Matlab is a powerful tool for controlling hardware tests in the lab. It is very popular with physicists. It is very useful for data-acquisition and analysis. As such xSIF provides a full Matlab API to SIF pods and slaves.

# 3 16-IDC SIF Interface

## 3.1 Pinout

| *Pin* | *Name* | *Type* | *Pod Pad* | *Slave Pad ZPAD* | *Description* |
|---|---|---|---|---|---|
| 1 | SIF_VDD | Power | - | - | Power supply used by SIF Slaves.<br>Configured by Pod jumper or computer software to be one of :<br>• Power from Pod to Slave PCB<br>• Power from Slave PCB to Pod |
| 2 | SIF_MISO | SIF | Input | Tristate Output<br>ZT_5 | MasterIn, SlaveOut data signal :<br>• Clocked out of Slave on posedge SIF_CLK<br>• Clocked into Master at any time in window (from negedge SIF_CLK to just before posedge SIF_CLK). |
| 3 | SIF_CS0 | IO | Bidir | Input<br>ZI_TS_PD | General purpose IO.<br>Recommended use :<br>• Chip Select 0 (Active High)<br>• Pod Output<br>• Slave Input<br>• Sampled by Slave system clk |
| 4 | SIF_MOSI | SIF | Output | Input<br>ZI_TS_PD | MasterOut, Slave In data signal :<br>• Clocked out of Master on posedge SIF_CLK<br>• Clocked into Slave on negedge SIF_CLK |
| 5 | GND | Power | - | - | Pod ground connected to Slave PCB ground by ribbon cable.<br>All signal voltages are relative to GND. |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | • Pin 5 connected to Pin 7 in Slave PCB<br>• Pin 5 connected to Pin 7 in Pod. |
| 6 | SIF_CLK | SIF | Output | Input<br><br>ZI_TS_PD | Clock signal for SIF_MOSI and SIF_MISO :<br>• Asynchronous to Slave system clk |
| 7 | GND | Power | - | - | Pod ground connected to Slave PCB ground by ribbon cable.<br>All signal voltages are relative to GND.<br>• Pin 5 connected to Pin 7 in Slave PCB.<br>• Pin 5 connected to Pin 7 in Pod. |
| 8 | SIF_CS1 | IO | Bidir | Input<br><br>ZI_TS_PD | General purpose IO.<br>Recommended use :<br>• Chip Select 1 (Active High)<br>• Pod Output<br>• Slave Input<br>• Sampled by Slave system clk |
| 9 | SIF_LOADB | SIF | Open Drain Bidir | Open Drain Bidir<br><br>ZB_TS_N_PU | Handshake signal for SIF Instructions (Normal mode SIF) :<br>• Normally high<br>• Active low<br>• Pulled low by Master then by Slave until SIF Instruction is completed<br>• Sampled by Slave system clk<br>• Changed on Slave system clk |
| 10 | SIF_CS2 | IO | Bidir | Input | General purpose IO.<br>Recommended use : |

| | | | | | |
|---|---|---|---|---|---|
| | | | | ZI_TS_ PD | • Chip Select 2 (Active High)<br>• Pod Output<br>• Slave Input<br>• Sampled by Slave system clk |
| 11 | SIF_CS3 | IO | Bidir | Input<br><br>ZI_TS_ PD | General purpose IO.<br>Recommended use :<br>• Chip Select 3 (Active High)<br>• Pod Output<br>• Slave Input<br>• Sampled by Slave system clk |
| 12 | SIF_REQ | IO | Bidir | - | General purpose IO.<br>Recommended use :<br>• Request from Pod to PCB Master in Multi-Master SIF systems.<br>• Pod Output<br>• PCB Master Input |
| 13 | SIF_ACK | IO | Bidir | - | General purpose IO.<br>Recommended use :<br>• Acknowledge from PCB Master to Pod in Multi-Master SIF systems.<br>• PCB Master Output<br>• Pod Input |
| 14 | SIF_RST | IO | Bidir | Input<br><br>ZI_TS_ PD | General purpose IO.<br>Recommended use :<br>• Asynchronous reset signal (active High)<br>• Pod Output<br>• Slave Input |
| 15 | STOPB | XAP1 | Bidir | Bidir | General purpose IO. |

| | | | | | |
|---|---|---|---|---|---|
| | | IO | | ZB_TS_N_PU | Recommended use :<br>• XAP1 debug signal. Special dedicated hardware (see section 9.3.1).<br>• Pod Bidirectional<br>• Slave Bidirectional |
| 16 | RUN_STEP | XAP1 IO | Bidir | Input<br><br>ZI_TS_PU | General purpose IO.<br>Recommended use :<br>• XAP1 debug signal. Special dedicated hardware (see section 9.3.1).<br>• Pod Output<br>• Slave Input |

All of the following diagrams on buffering, protection and termination (for Pods and Slave PCBs) assume that the General Purpose IO pins are all being used in their recommended directions (as shown in the above table).

### 3.1.1 SIF Slave Pad Types

The above table refers to various ZPADs for the SIF Slave. The characteristics of these pads are :

| *ZPAD* | *Type* | *Description* |
|---|---|---|
| ZI_TS_PD | TTL Input with Pulldown | TTL Threshold (0.8V, 2.0V)<br>Schmitt Trigger<br>5V Tolerant<br>Pulldown resistor (10 kohm ?) |
| ZI_TS_PU | TTL Input with Pullup | TTL Threshold (0.8V, 2.0V)<br>Schmitt Trigger<br>5V Tolerant<br>Pullup resistor (10 kohm ?) |
| ZT_5 | CMOS Tristate Output | 8 mA Sink and Source<br>Drive to 0V or SIF_VDD or high-impedance<br>Slew-rate control |
| ZB_TS_N_PU | TTL Open-Drain Bidirectional with Pullup | TTL Threshold (0.8V, 2.0V)<br>Schmitt Trigger<br>5V Tolerant<br>6 mA Sink<br>Pullup resistor (10 kohm ?)<br>Drive to 0V or resistive to SIF_VDD<br>Slew-rate control |

### 3.2 Connector

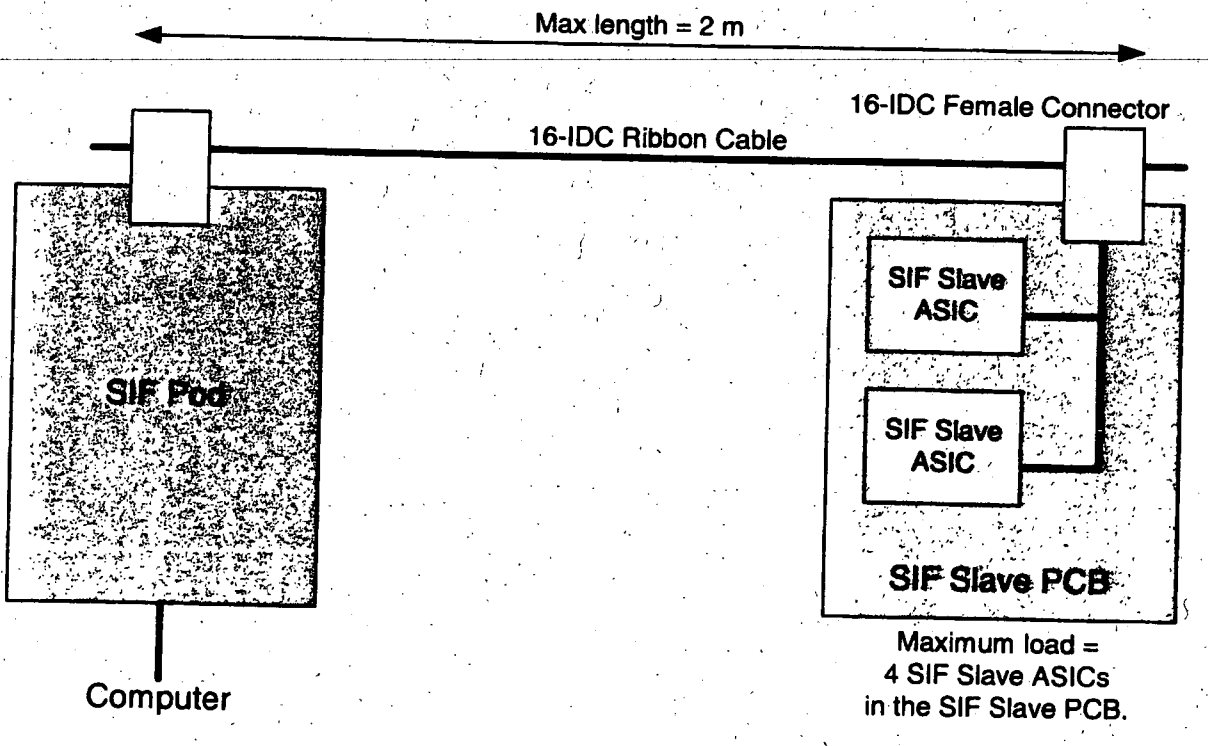
Standard bump-polarised 16-IDC connector :

- 2 * 8 pins
- 2.54 mm pitch
- Cable connectors = female
- PCB connectors (on SIF Pod and SIF Slave PCB) = male

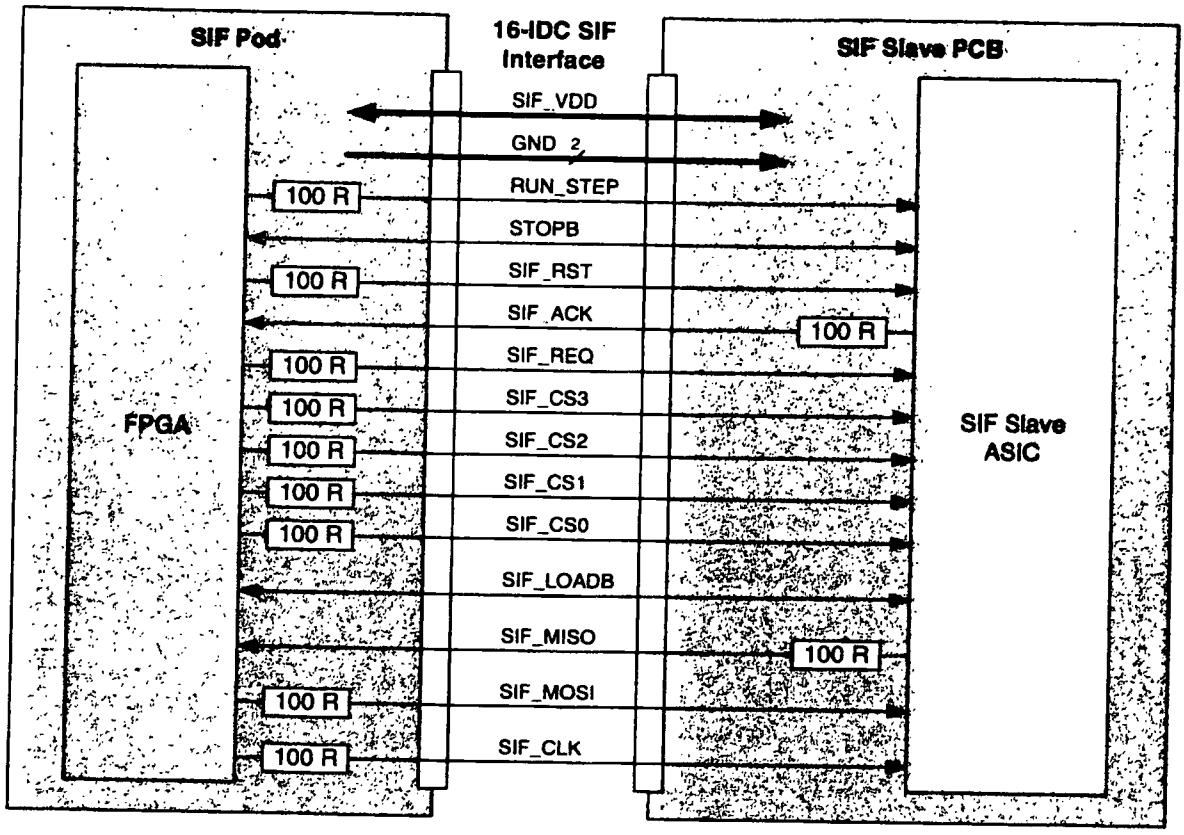
### 3.3 Cable

The cable should be standard 16-IDC ribbon cable :

- 1.27 mm pitch.
- Straight cable or twisted pair (e.g Twist and Flat from 3M or Spectra Strip).
- 100 ohm impedance.
- Max Length = 2 m.
- Max speed (bits per second) decreases as cable length increases.
- Can have several female connectors along length of cable. Pod at one end. Termination block at other end. Slaves in between.



## 3.4 Termination




The above diagram shows the termination strategy used with the 16-IDC SIF Interface. More details on the actual complete circuits that should be used in the Slave PCB and Pod are given in sections 8 and 9. In practise these circuits have to combine the requirements for termination and ESD/High Voltage protection.

The termination strategy is to have 100 ohm series source termination at each driver output, as shown in the diagram above. This is to match the standard impedance of the ribbon cable (which is 100 ohm).

This works well for point-to-point connections (i.e a single Slave, as shown above). Any wave reflections from the destination should be perfectly absorbed in the 100 ohm series resistor at the source (thereby preventing multiple reflections).

However, it does not work so well when there are multiple Slaves in the Slave PCB (i.e 2 to 4). The reflections will be seen by the intermediate Slaves which could cause errors. One way to deal with this is to use slower edge speeds and shift frequencies. Another is to add buffers to the Slave PCB so that the cable does only see point-to-point loads. This second method is recommended and shown in section 8 on SIF Slave PCBs.

In practise, the SIF Pod has ESD/High voltage protection on every signal anyway (see later section on SIF Master). This includes a 100 series resistor. So in fact this resistor can double up as protection and source series termination for the Pod outputs. Actually,

the Pod contains protection circuitry for the Inputs (SIF_MISO, SIF_ACK) & Bidirectionals (SIF_LOADB, STOPB) as well. This is not ideal but does not do much harm (as the gate inputs are such high impedance anyway).

It is necessary for the Slave PCB to have 100 ohm source series termination close to the SIF_MISO and SIF_ACK pins on each SIF Slave. It is best if the Slaves are positioned as close as possible to the 16-IDC SIF connector on the Slave PCB.

## 3.5 Maximum Shift Frequencies

The maximum frequency for SIF_CLK, SIF_MOSI and SIF_MOSI depends upon :

- Maximum Pod shift frequency
- Whether Slaves are buffered or not in the Slave PCB
- Length of 16-IDC ribbon cable
- Number of Slaves in the Slave PCB
- Design of the SIF Slave PCB (termination & position of SIF Slaves relative to the 16-IDC SIF connector)

### 3.5.1 Un-buffered Slave PCB (Production)

| *Estimated Maximum Shift Frequency (MHz)* | | *Number of un-buffered Slaves in Slave PCB* | | | |
|---|---|---|---|---|---|
| | | *1* | *2* | *3* | *4* |
| *Length of 16-IDC Ribbon Cable (mm)* | *50* | 25 | 10 | 10 | 10 |
| | *100* | 20 | 5 | 5 | 5 |
| | *300* | 15 | 1 | 1 | 1 |
| | *1000* | 10 | - | - | - |
| | *2000* | 5 | - | - | - |

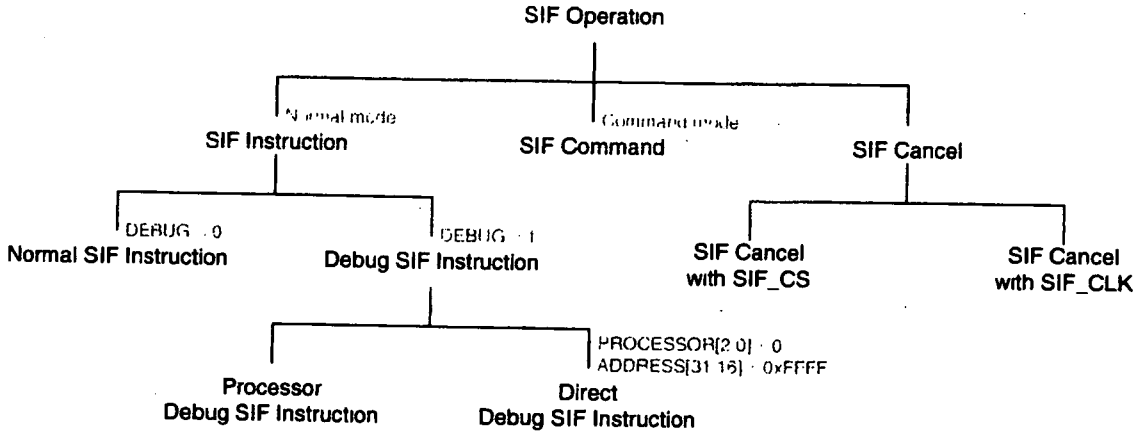
### 3.5.2 Buffered Slave PCB (Test)

| *Estimated Maximum Shift Frequency (MHz)* | | *Number of buffered Slaves in Slave PCB* | | | |
|---|---|---|---|---|---|
| | | *1* | *2* | *3* | *4* |
| *Length of 16-IDC Ribbon Cable (mm)* | *50* | 50 | 20 | 20 | 20 |
| | *100* | 40 | 10 | 10 | 10 |
| | *300* | 30 | 5 | 5 | 5 |
| | *1000* | 20 | 2 | 2 | 2 |
| | *2000* | 10 | 1 | 1 | 1 |

# 4 SIF Slave Protocol

The SIF Slave protocol describes the way a SIF Master (e.g a Pod) must control the SIF signals (SIF_CS, SIF_CLK, SIF_MOSI, SIF_MISO, SIF_LOADB) in order to read from and write to a SIF Slave (ASIC or FPGA).

SIF Operations can be categorised as follows :




There are 2 modes for SIF Operations :

- SIF Instructions in Normal mode
- SIF Commands in Command mode

## 4.1 Pins used for Normal and Command modes

A SIF Slave is either in Normal or Command mode. At reset it is in Normal mode. Most of the time, the SIF Slave is used in Normal mode.

A full SIF Slave has 5 external SIF pins (SIF_CS, SIF_LOADB, SIF_CLK, SIF_MOSI, SIF_MISO).

If the SIF Slave will never be used in multi-slave networks, then it only needs to have 4 external SIF pins. In such cases SIF_CS should be tied high (= 1) inside the slave.

To access a SIF Slave, SIF_CS should be 1 (for Normal and Command Modes). However, SIF_CS only affects the SIF_LOADB and SIF_MISO pins in a SIF Slave.

SIF_CS does not affect the 2 input pins to the shift register (SIF_CLK, SIF_MOSI). Data is always fed into the shift register, even if SIF_CS = 0.

SIF_MISO is tri-state when SIF_CS = 0. SIF_MISO is enabled when SIF_CS = 1.

When SIF_CS = 0, the internal copy of SIF_LOADB is forced to 1 (meaning that it is not possible to do a SIF Read or Write).

Normal mode uses all 5 SIF pins and supports variable length shift registers (from one ASIC design to another).

In effect, Command mode only uses 4 SIF Pins (SIF_LOADB not used). When in Command mode, the shift register is treated as fixed length (across all designs). 8-bit Commands are fed into SIF_MOSI and 8-bit Responses are generated from SIF_MISO.

## 4.2 Bit Order

All SIF Operations shift data MSB (most significant bit) first :

- From Master to Slave on SIF_MOSI
- From Slave to Master on SIF_MISO
- All SIF Instructions in Normal mode
- All fields in Normal mode (Address, Control, Data, Status)
- All SIF Commands in Command mode
- All fields in Command mode (Command, Response)

## 4.3 SIF Pod Timing

The SIF Pod must obey various timing constraints on the signals it generates :

- SIF_CS
- SIF_CLK
- SIF_MOSI
- SIF_LOADB_MASTER

The SIF Toolkit on the computer allows the user to set 2 times :

- SIF_CLK period (ns)
- SIF_LOADB period (number of SIF_CLK periods). This is the pulse time that the SIF Pod holds SIF_LOADB low for.

The limits on these times vary from one SIF Pod to another. SIF Pods must obey these limits when performing the various SIF Operations described in the rest of this section.

### *4.3.1 SIF_CLK Duty Cycle*

In between SIF Operations, the SIF Pod holds SIF_CLK low.

A SIF_CLK cycle is therefore defined as being in the following order :

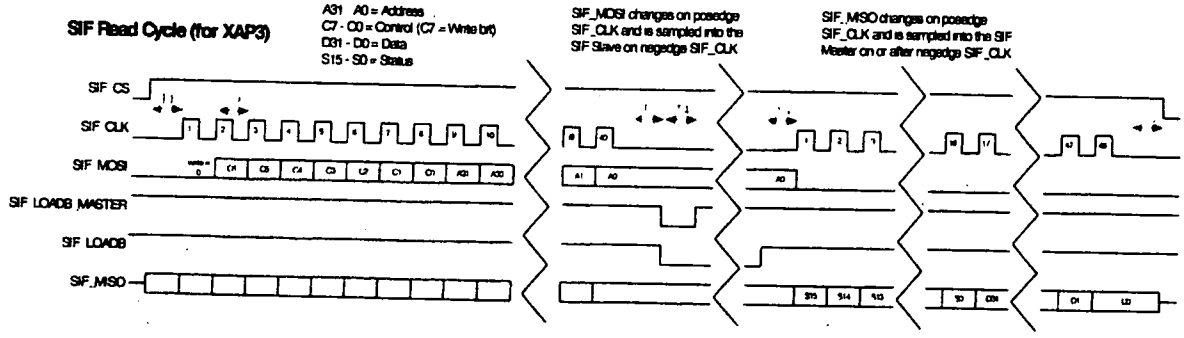
- SIF Pod first pulls SIF_CLK high for Tclkhigh (changes SIF_MOSI on posedge)
- SIF Pod then pulls SIF_CLK low for Tclklow (samples SIF_MISO on negedge)

The duty-cycle on SIF_CLK for Tclkhigh:Tclklow must be between :

- 60:40
- 40:60

## 4.4 SIF Read

### *4.4.1 Single SIF Read*



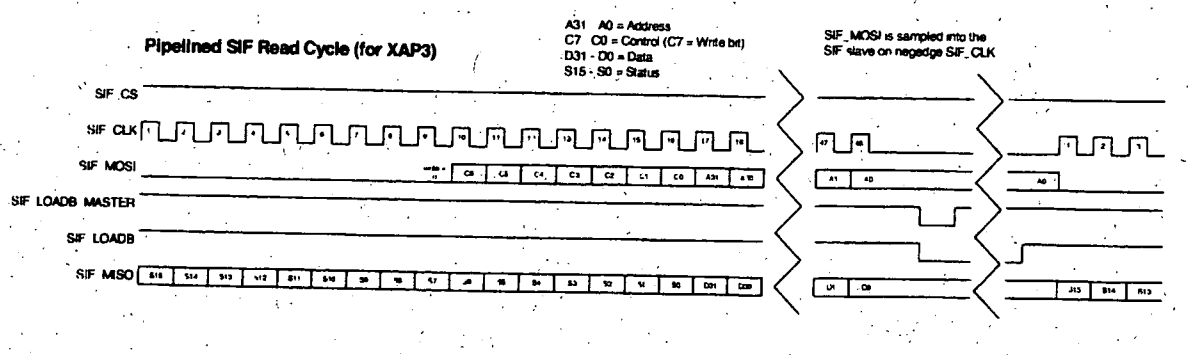

The SIF Read sequence is :

- SIF Master shifts Control then Address to SIF Slave on SIF_MOSI.
- SIF Master pulls SIF_LOADB low for short pulse.
- SIF Slave holds SIF_LOADB low until SIF Read is complete.
- SIF Master shifts Status then Data out from SIF Slave on SIF_MISO.

The SIF Pod must obey this timing table :

| *Name* | *Description* | *Min Time* |
|---|---|---|
| T1 | posedge SIF_CS to first posedge SIF_CLK | SIF_CLK period |
| T2 | posedge SIF_CLK to posedge SIF_CLK | SIF_CLK period |
| T3 | end SIF_CLK cycle (end Tclklow) to negedge SIF_LOADB_MASTER | SIF_CLK period |
| T4 | negedge SIF_LOADB_MASTER to posedge SIF_LOADB_MASTER | SIF_LOADB period |
| T5 | posedge SIF_LOADB to posedge SIF_CLK | SIF_CLK period |
| T6 | end SIF_CLK cycle (end Tclklow) to negedge SIF_CS | SIF_CLK period |

### 4.4.2 Pipelined SIF Reads




Pipelined SIF Read performs n SIF Reads by iterating the following cycle n times :

- SIF Master simultaneously shifts :
  - Control then Address to SIF Slave on SIF_MOSI
  - Status then Data (from previous Read) out from SIF Slave on SIF_MISO

  The length of the shift is determined by the longer of the above two.
- SIF Master pulls SIF_LOADB low for short pulse.
- SIF Slave holds SIF_LOADB low until SIF Read is complete.

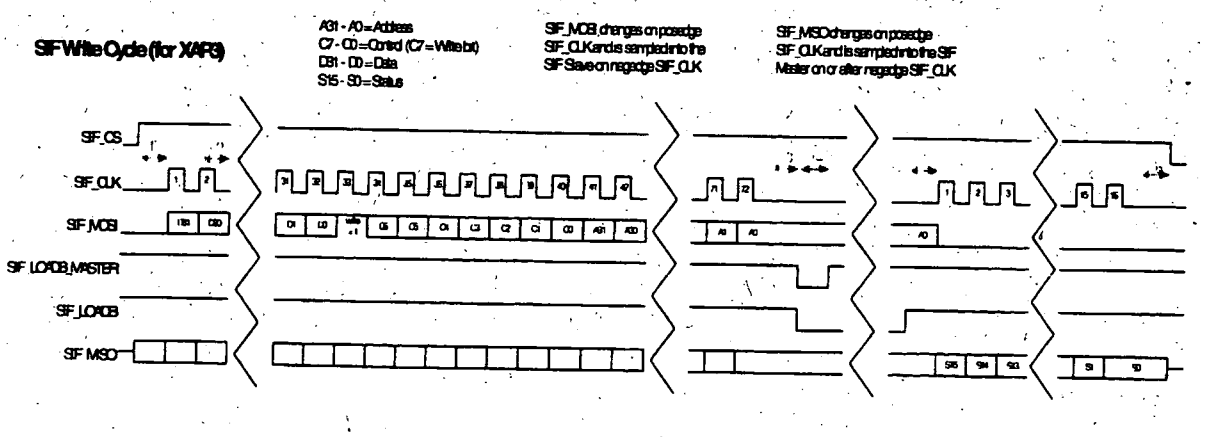
{Status, Data} from the first cycle are ignored.

{Status, Data} are shifted out from SIF Slave on SIF_MISO after the nth cycle.

The SIF Pod must obey the same timing constraints as described for SIF Read.

## 4.5 SIF Write

### 4.5.1 Single SIF Write



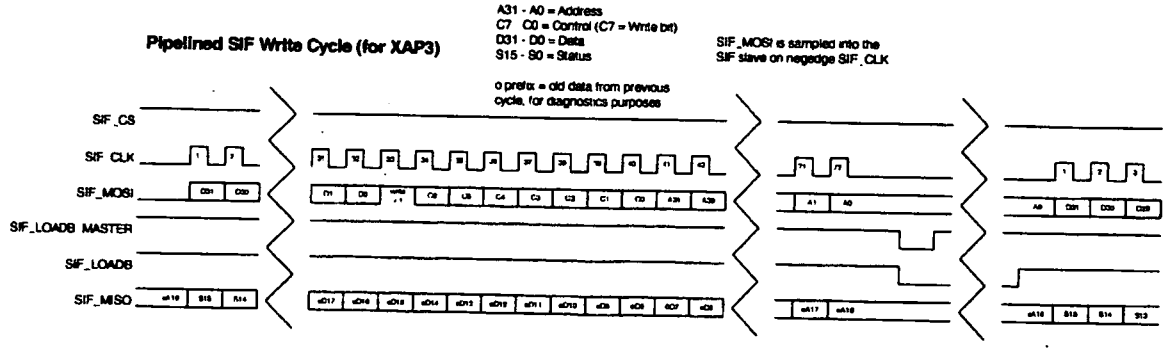

The SIF Write sequence is :

- SIF Master shifts Data then Control then Address to SIF Slave on SIF_MOSI.
- SIF Master pulls SIF_LOADB low for short pulse.
- SIF Slave holds SIF_LOADB low until SIF Write is complete.
- SIF Master shifts Status out from SIF Slave on SIF_MISO.

The SIF Pod must obey this timing table :

| *Name* | *Description* | *Min Time* |
|---|---|---|
| T1 | posedge SIF_CS to first posedge SIF_CLK | SIF_CLK period |
| T2 | posedge SIF_CLK to posedge SIF_CLK | SIF_CLK period |
| T3 | end SIF_CLK cycle (end Tclklow) to negedge SIF_LOADB_MASTER | SIF_CLK period |
| T4 | negedge SIF_LOADB_MASTER to posedge SIF_LOADB_MASTER | SIF_LOADB period |
| T5 | posedge SIF_LOADB to posedge SIF_CLK | SIF_CLK period |
| T6 | end SIF_CLK cycle (end Tclklow) to negedge SIF_CS | SIF_CLK period |

### 4.5.2 Pipelined SIF Writes




Pipelined SIF Write performs n SIF Writes by iterating the following cycle n times :

- SIF Master simultaneously shifts :
  - Data then Control then Address to SIF Slave on SIF_MOSI
  - Status (from previous Write) out from SIF Slave on SIF_MISO

  The length of the shift is determined by the longer of the above two.
- SIF Master pulls SIF_LOADB low for short pulse.

- SIF Slave holds SIF_LOADB low until SIF Write is complete.

Status from the first cycle is ignored.

Status is shifted out from SIF Slave on SIF_MISO after the nth cycle.

The SIF Pod must obey the same timing constraints as described for SIF Write.

### 4.6 Synchronising a FIFO with SIF_LOADB for Data Acquisition

SIF Instructions are always initiated by the SIF Master. The SIF Slave only ever responds. This can still be used for data acquisition tasks by using the Status field as a timestamp (e.g a counter where the LSB toggles at 1 to 100 kHz). The SIF Master can then reconstruct the acquired data, discarding duplicate samples and spotting if there are any missing samples.

For very fast data acquisition it can be simpler if the SIF Slave initiates the SIF transaction. The way to do this is as follows :

- SIF Slave contains a memory-mapped FIFO. The depth depends upon how much buffering the SIF Master requires. The FIFO can be read, but not written by the SIF. When SIF reads are made to this particular address, the SIF Slave can hold SIF_LOADB low until a valid new data sample is available. This is over and above the normal waits for a SIF or SLEEPSIF instructions when the selected SIF Processor is running.
- SIF Master continuously reads the FIFO (via the SIF). When the selected SIF Processor is running, the SIF reads will occur at the first SIF or SLEEPSIF instruction after valid data is available in the FIFO. When the selected SIF Processor is stopped the SIF read will occur immediately after valid data is available.
- SIF Slave writes data samples to the FIFO as they occur (regular frequency expected)
- If the SIF Master gets ahead of the SIF Slave, the SIF Slave will hold SIF_LOADB low until the next SIF or SLEEPSIF instruction (if running) after the next data sample is ready. This means that the SIF Master will never get any duplicate copies of data samples.
- If the SIF Master falls behind the SIF Slave, the FIFO will get corrupted. The SIF Slave indicates this to the SIF Master by setting an overflow bit in the Status field.

This feature is application specific. Not all SIF Slaves will contain such a FIFO. SIF Slaves which do contain such a FIFO can hold SIF_LOADB for longer, when the SIF Master makes a SIF Read to the FIFO. However, SIF reads to any other address will behave as normal (without any stretching of SIF_LOADB).

## 4.7 SIF Cancel

If the SIF Master issues a SIF Instruction to a selected SIF Processor inside the Slave that cannot be completed, then the SIF Slave will continue to hold SIF_LOADB low for ever. This can happen for a variety of reasons, including :

- SIF Processor is running (in Normal or Debug mode). SIF Instruction requires a SIF or SLEEPSIF instruction, but there aren't any in the processor code.

The SIF Master must be able to detect that SIF_LOADB is stuck low and timeout after a specified period. The SIF is then locked and needs to be cleared by issuing a SIF Cancel instruction. This can be done using one of the following 2 methods.

When the SIF Master issues a SIF Cancel to the SIF Slave, it cannot be certain whether the previous SIF Instruction (read or write) will be executed or not. The Slave might execute the SIF Instruction in the period between the Master transmitting the start of the SIF Cancel and the Slave receiving the end of the SIF Cancel. However, the Master can assume that if the SIF Instruction is executed, that it will be exactly as requested. The Master doesn't know whether the read or write has occurred, but it does know that if it did occur, then it was executed correctly. The Slave will never execute any unrequested SIF Instructions (reads or writes during SIF Cancel or any other period).

### *4.7.1 SIF Cancel with SIF_CS*

If the SIF Master pulls SIF_CS low (sampled by Slave system clock), the SIF Slave will no longer be selected. The SIF Slave will stop pulling SIF_LOADB low, allowing the signal to go high again. It does not matter what the values are on SIF_CLK, SIF_MOSI or SIF_LOADB. A SIF Slave cannot hold SIF_LOADB low if its SIF_CS pin is low.

Some SIF Slaves do not use the SIF_CS pin (i.e can only be used in Single Slave networks). In such devices, the SIF_CS signal is permanently tied high inside the chip. Therefore the method for clearing a locked SIF cannot be used in all SIF Slaves.

This feature is only implemented in SIF Slaves from 2005 onwards. It is implemented in the CVS Verilog repository for sif and xsif. It is implemented in XAP2 and XAP3. It is not implemented in XAP1.

### *4.7.2 SIF Cancel with SIF_CLK*

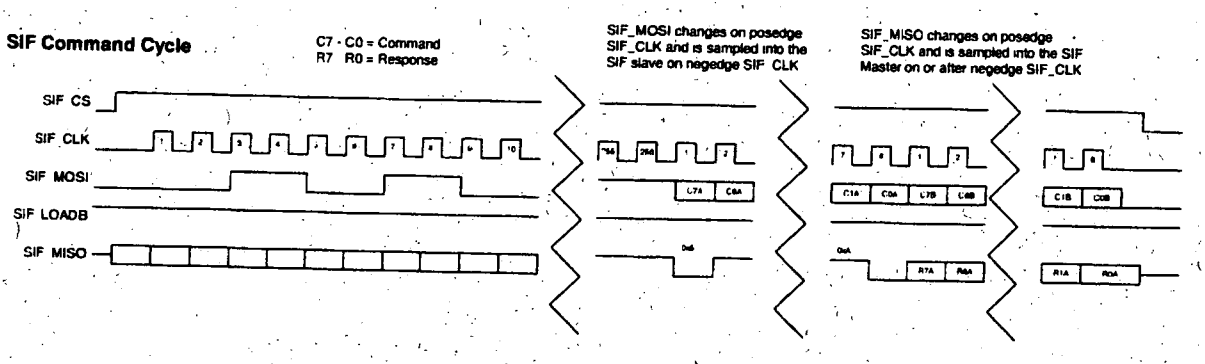
If the SIF Master issues 32 cycles (high then low) on SIF_CLK, then on the 32$^{nd}$ negedge the SIF Slave will stop pulling SIF_LOADB low, allowing the signal to go high again.

It does not matter what the values are on SIF_CS, SIF_MOSI or SIF_LOADB during this sequence. All SIF Slaves on the network will release SIF_LOADB after the 32$^{nd}$ negedge on SIF_CLK. Of course, this does not cause any change of state in SIF Slaves that were not holding SIF_LOADB low.

The reason this protocol works is because there should not be any cycles on SIF_CLK when SIF_LOADB is low. In fact SIF_CLK should always be low when SIF_LOADB is low.

This feature is implemented in SIF devices from 1999 onwards. It is implemented in the CVS Verilog repository for sif and xsif. It is implemented in XAP2 and XAP3. It is not implemented in XAP1.

## 4.8 SIF Command




The maximum frequency for SIF_CLK in Command mode is 1 MHz. This includes the 256 clock sequence to transfer from Normal to SIF Mode.

At reset, the SIF Slave is in Normal mode. To use SIF Commands, the SIF Slave must first be put into Command mode. This is done by raising the SIF Command Mode Counter value to 256, which is done as follows :

- SIF_CS = 1. Sampled by Slave system clock.
- SIF_LOADB = 1 (not used). Sampled by Slave system clock.
- 256 cycles of SIF_CLK.
- SIF_MOSI = 0x33333.... (sampled on negedge SIF_CLK)

After the 256$^{th}$ negedge on SIF_CLK, SIF goes in to Command mode which means :

- SIF_MISO is not connected to STATUS[msb], but to the Response output instead.
- SIF_LOADB is not used.
- SIF_MOSI will be interpreted as fixed 8-bit Commands (msb first, sampled on negedge SIF_CLK).
- SIF_MISO will generate fixed 8-bit Responses (msb first, changes on posedge SIF_CLK) to the previous command. Responses should be treated as unsigned numbers from 0x00 to 0xFE. 0xFF is used as the Response to any unrecognised Command.

Note that the SIF Slave only has Tclklow (2nd half of SIF_CLK cycle) from entering Command mode (negedge SIF_CLK) to putting Response[7] on SIF_MISO (posedge SIF_CLK). With a 1 MHz SIF_CLK this is 500 ns, and could be as low as 400 ns with a 40:60 duty cycle on SIF_CLK. This is the why the fastest frequency for Command mode is only 1 MHz.

The SIF Command Mode Counter goes back to 0 (thus putting the SIF back into Normal Mode) when any of the following occur :

- SIF Command = 0x00 (i.e SIF_MOSI = 0, for 8 cycles of SIF_CLK)
- SIF_LOADB = 0 (even if still in Normal Mode). Sampled by Slave system clock.
- SIF_CS = 0 (even if still in Normal Mode). Sampled by Slave system clock.

SIF Commands and Responses are always 8-bit. The Command is fed into the SIF Slave on SIF_MOSI, while the Response from the previous Command is simultaneously fed out on SIF_MISO. While the first Command is fed into SIF_MOSI, 0x5A is output on SIF_MISO. This confirms to the Master, that the Slave has successfully entered Command mode.

All of the SIF Commands defined so far are 'Get' functions. They all query the SIF configuration and in effect read a fixed ROM inside the SIF. They do not update the SIF Slave in any way. In future there may be some simple 'Set' functions for general IO lines inside the SIF. SIF Commands cannot be used to read or write any processor or memory address space in the SIF Slave. That can only be done by Normal mode SIF Instructions.

The SIF Slave must respond to the following SIF Commands :

| *OpCode* | *Command* | *Response* | *XAP3 Response* |
|---|---|---|---|
| 0x00 | Return to Normal Mode | End of Command Mode | - |
| 0x01 | Confirm synchronisation | 0x5A | 0x5A |
| 0x02 | SIF Type | 0 = Undefined<br>1 = 36-bit standard XAP1<br>2 = 64-bit standard XAP2<br>3 = 88-bit standard XAP3<br>4 = 52-bit standard XAP4 | 3 |
| 0x10 | Chip ID – byte 0<br>New Response added for each new chip. | 0 = Undefined<br>1 = 1st Chip ID<br>0xFD = 253rd Chip ID<br>0xFE = Use Command 0x11 | 0 |

| | | | |
|---|---|---|---|
| 0x11 | Chip ID - byte 1<br>Use this when all Responses to Command 0x10 have been used. | 0 = 254th Chip ID<br>1= 255th Chip ID<br>0xFE = Use Command 0x12 | NA<br>i.e 0xFF |
| 0x20 | Address Field Length | 1 to 32 | 32 |
| 0x30 | Control Field Length | 1 to 32<br>Includes Write Bit | 8 |
| 0x31 | Number of PROG_DATAB bits | 0 = Von Neumann<br>1 = Harvard architecture | 0 |
| 0x32 | Number of PROCESSOR bits | 0 to 8 | 3 |
| 0x33 | Number of DEBUG bits | 0 to 1 | 1 |
| 0x34 | Number of SIZE bits | 0 to 2 | 2 |
| 0x35 | Number of PARITY_AC bits | 0 to 1 | 1 |
| 0x36 | Write Bit Polarity | 1 = Active High Write<br>0 = Active Low Write | 1 |
| 0x40 | Data Field Length | 1 to 32 | 32 |
| 0x50 | Status Field Length | 0 to 32 | 16 |
| 0x51 | Number of STAT bits | 0 to 32 | 8 |
| 0x52 | Number of ERROR_CODE bits | 0 to 8 | 4 |
| 0x53 | Number of ERROR_TYPE bits | 0 to 1 | 1 |
| 0x54 | Number of ERROR bits | 0 to 1 | 1 |
| 0x55 | Number of PARITY_D bits | 0 to 1 | 1 |
| 0x56 | Number of PARITY_S bits | 0 to 1 | 1 |
| 0x60 | SIF Processor 0 | 0 = No Processor<br>1 = XAP1<br>2 = XAP2<br>3 = XAP3 | 3 |

| | | | |
|---|---|---|---|
| | | 4 = XAP4<br>0x11 = APE1<br>0x12 = APE2 | |
| 0x61 | SIF Processor 1 | Response Values as above. | 0 |
| 0x62 | SIF Processor 2 | Response Values as above. | 0 |
| 0x63 | SIF Processor 3 | Response Values as above. | 0 |
| 0x64 | SIF Processor 4 | Response Values as above. | 0 |
| 0x65 | SIF Processor 5 | Response Values as above. | 0 |
| 0x66 | SIF Processor 6 | Response Values as above. | 0 |
| 0x67 | SIF Processor 7 | Response Values as above. | 0 |
| Any other OpCode | Unrecognised Command | 0xFF | 0xFF |

xIDE should use the following sequence to identify the chip and its SIF configuration :

| *Step* | *Command* | *Response != 0* | *Response = 0* |
|---|---|---|---|
| 1 | 0x10 | Chip ID defines everything about this particular chip design.<br>This should be used for ASIC designs. | Chip ID - not defined.<br>Proceed to Step 2. |
| 2 | 0x02 | SIF Type defines enough for xIDE to work with the chip.<br>Proceed to Step 3.<br>This should be used for FPGAs and demonstrators. | SIF Type - not defined.<br>Proceed to Step 4. |
| 3 | 0x60-0x67 | Identifies the SIF Processors inside the chip. | No SIF Processor at this ID. |
| 4 | 0x20-0x5F | Identifies the details for Address, Control, Data and Status fields for non-standard SIF Types.<br>This should be used for ASICs or FPGAs designed by 3rd parties. | |

If Command 0x10 (Chip ID) is defined, then Commands 0x20-0x67 are optional.

If Command 0x02 (SIF Type) is defined, then Commands 0x20-0x5F are optional.

It is best if 0x02 is always defined, even when 0x10 is defined.

Commands 0x00, 0x01, 0x02, 0x10 should always be implemented (Response != 0xFF).

# 5 SIF Error Processing

## 5.1 Data Integrity

### 5.1.1 Data Integrity between Computer and Pod

There will not be any data corruption between Computer and Pod because the SIF Pod Protocol includes a CRC check on each packet. This is used for all SIF Pods apart from the LPT Pod. In addition, Ethernet and USB protocols contain their own error-correction protocols (by error-checking and re-send) to guarantee data transfer without corruption.

There could be data corruption between Computer and the LPT Pod, because it does not implement the SIF Pod Protocol and therefore does not contain any error-correction protocol.

### 5.1.2 Data Integrity between Pod and Slave

XAP3 SIF devices include hardware parity checking between the Master and Slave (more detail later).

Older devices such as XAP2 and XAP1 do not provide any error checking. In such devices there could be data corruption on the serial link between the SIF Pod (Master) and Slave (using SIF_CLK, SIF_MOSI, SIF_MISO over the 16-IDC ribbon interface).

It is safe to assume that the parallel loads inside the Slave between the SIF shift register and the memory bus will function correctly. This part of the Slave circuit will be correct by design. It will not change from one installation or Pod to another.

This means that if data integrity is maintained over the 16-IDC ribbon serial link, then there will not be any read or write errors between the Pod and Slave. This data-integrity may vary through time. For example, interference from switching motors or relays could cause data errors.

## 5.2 Error Correction

SIF does include error-correction protocols between the Computer and Pod.

SIF does not include any error-correction between the Pod and Slave. However, the parity-checking does mean that single-bit corrupted SIF Instructions do not get executed. Such events are reported back to the Master by the error bits in the Status field.

## 5.3 Error Detection

A variety of error-detection methods are used by SIF. Some are automatically implemented in the xSIF drivers (parity-checking and diagnostics). Others must be implemented at the software application level (e.g CRC or checksum across large packets).

### *5.3.1 Parity Checking*

XAP3 SIF devices include parity checking between the Master and Slave. This will expose single-bit, stuck-at-0 and stuck-at-1 faults on :

- SIF_MOSI from Master to Slave (stuck-at faults exposed by PARITY_AC)
- SIF_MISO from Slave to Master (stuck-at faults exposed by Parity_S)

The Slave will not execute a SIF Instruction if it finds a parity error in PARITY_AC (on the Address and Control fields). It will report this and other errors in the Status field returned to the Master. This means that single-bit corrupted SIF Reads will not be executed. Corrupted SIF Writes will be executed if the error is in the Data field (which is parity protected as an output, but not as an input). However, the Master will check the PARITY_D bit in the Status field, which will indicate that the data was corrupted. This tells the Master about the corrupt SIF Write after the event has taken place. The Master must then take corrective action.

The Master will always check the PARITY_D (on Data field) and PARITY_S (on Status field) bits in the Status field. This will tell the Master if the data transferred from the Slave has been corrupted (exposing all single-bit errors).

### *5.3.2 Data Integrity in Shift Registers and Cable*

SIF computer applications can and should run diagnostics checks on the integrity of the serial links from Pod to Slave and back to Pod. This is possible because the SIF Slave shift register is continuous from the SIF_MOSI input to the SIF_MISO output (like JTAG, but unlike SPI and $I^2C$). This means it is possible to check that the data sent from Computer to Slave (via Pod) is the same as that received back from Slave to Computer. This is best if SIF_LOADB remains high (i.e no read or write is performed). The SIF_CLK frequency can be increased until this test fails. This reveals the maximum frequency that the serial link can safely be run at.

Such checks can be done by the application software. They are also done automatically by the xSIF drivers when the appropriate diagnostics level is selected.

### 5.3.3 Diagnostics

The Computer can run data integrity checks on some or all (affects speed) of the data during SIF reads and writes. These checks are in addition to the parity-checks that always operate in the XAP3 SIF. The data integrity checks are performed by the driver when diagnostics is turned on (level 1 = default case for all software applications). Thus the application software can continuously monitor the quality of the serial link between Pod and Slave during SIF operation. This is helpful in exposing hardware problems between the Pod and Slave. SIF Diagnostics can be set to different levels :

- 0 Off
- 1 Run data integrity checks on all single SIF Reads and Writes (default setting)
- 2 Run data integrity checks on all single and multiple SIF Reads and Writes

## 5.4 User view of SIF error-handling

The SIF system (as seen from the controlling computer) will tell the user when anything is broken and make suggestions as to what it might be.

The SIF Slave does not perform any error-correction as this is too big an overhead. However, the SIF Pod and drivers do run automatic parity-checks (in XAP3 SIF) and diagnostic-checks (in all SIF devices when the Diagnostics level is set appropriately).

The SIF Pod does perform error-correction on the link to the computer, by using the CRC in each packet of the SIF Pod Protocol.

# 6 SIF Networks

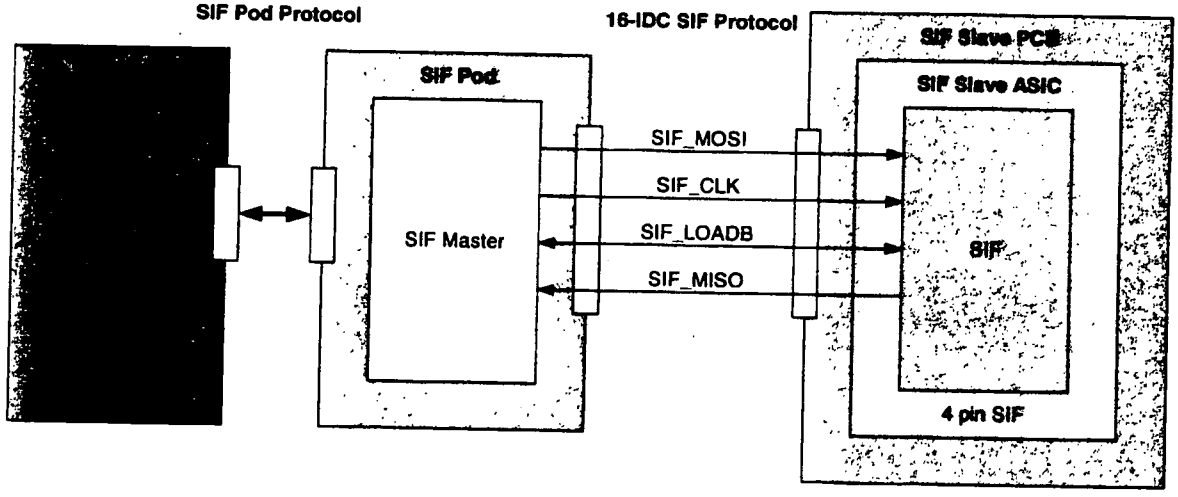
The following sections show the 3 kinds of network you can have from a single SIF Pod. Even bigger networks can be built by using multiple SIF Pods. This is particularly powerful with Ethernet SIF Pods which can be physically located a long distance from the controlling computer. By using an Internet link the computer can even control SIF Slaves in remote countries.

It is possible to have multiple Computers on the network. However, at any one time :

- A SIF Slave can only be controlled by one Computer
- A SIF Pod can only be controlled by one Computer

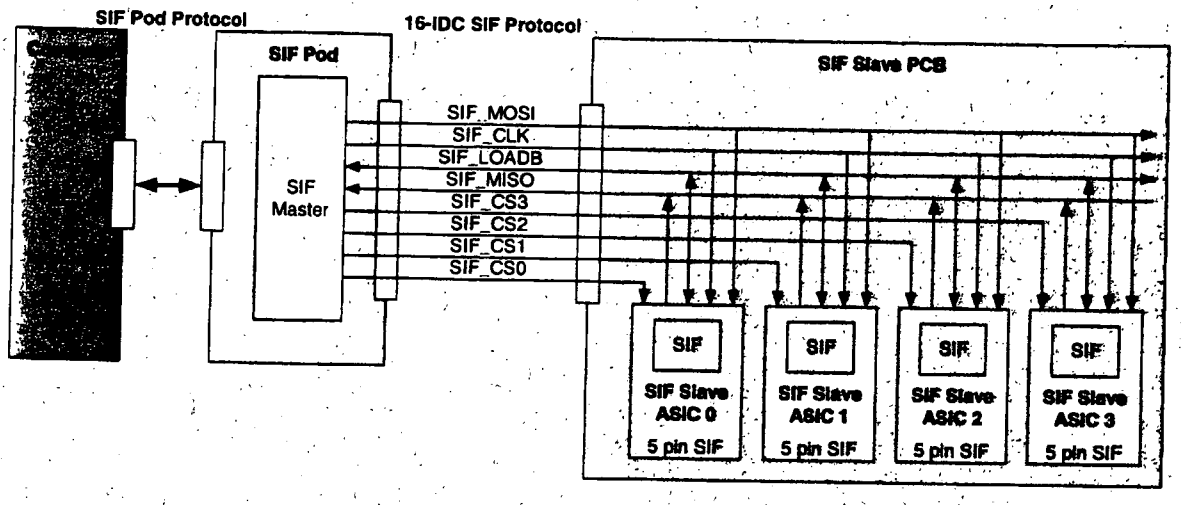
After the 3 network types, there are descriptions of the control and timing that must exist between the Master and its Slaves.

## 6.1 Single Master, Single Slave




Without SIF_CS you can only have 1 SIF Master and 1 SIF Slave. This means that you can only have one SIF Slave per SIF Pod.

## 6.2 Single Master, Multi Slave




If the SIF Slave ASICs have a SIF_CS input, you can have a network with 1 SIF Master (e.g SIF Pod) and multiple SIF Slaves. The Master controls the SIF_CS inputs to each SIF Slave. SIF_MISO from each Slave is tristated when the associated SIF_CS input is low. SIF_MISO is enabled when SIF_CS is high. Thus the Master should never pull more than one SIF_CS line high at a time (or there will be contention on the SIF_MISO line).

When SIF_CS is low it does 2 things inside the Slave ASIC :

- Tristates SIF_MISO
- Disables SIF_LOADB

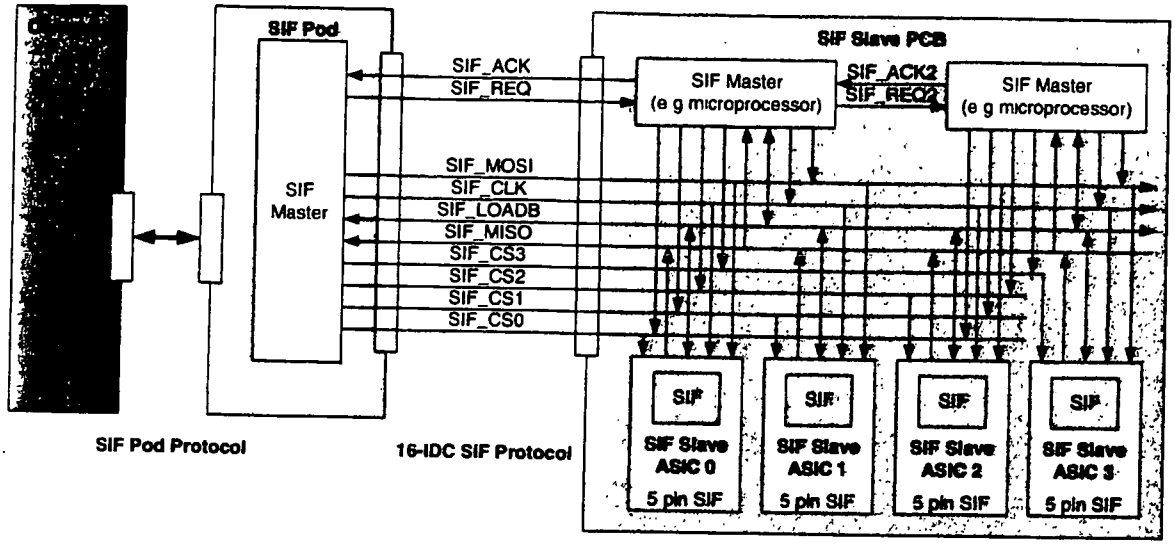
It does not affect SIF_CLK or SIF_MOSI in any way. It is still possible to clock new data into the SIF Shift Register even if SIF_CS is low. However, SIF_MISO will be tristated so this output will be high impedance even if the SIF Shift Register is clocked.

If SIF_CS is low, the Slave will ignore SIF_LOADB when it is pulled low by the Master. The Slave will not latch SIF_LOADB low and will not queue up a SIF Transfer for the next SIF cycle.

When SIF_CS is high it enables the SIF inside the Slave ASIC :

- Enables SIF_MISO
- Enables SIF_LOADB

## 6.3 Multi Master, Multi Slave




This configuration is not often used. It is useful when the end product contains a microprocessor and a SIF Slave ASIC. In this situation, the SIF Master can be the microprocessor or the SIF Pod. It allows the SIF Pod to negotiate with the microprocessor to decide who should have control of the SIF (thus enabling communication with the SIF Slaves). This is done with 2 signals :

- SIF_REQ (request from Pod to Microprocessor)
- SIF_ACK (acknowledge from microprocessor to Pod)

When the SIF Pod wants to control of the SIF it :

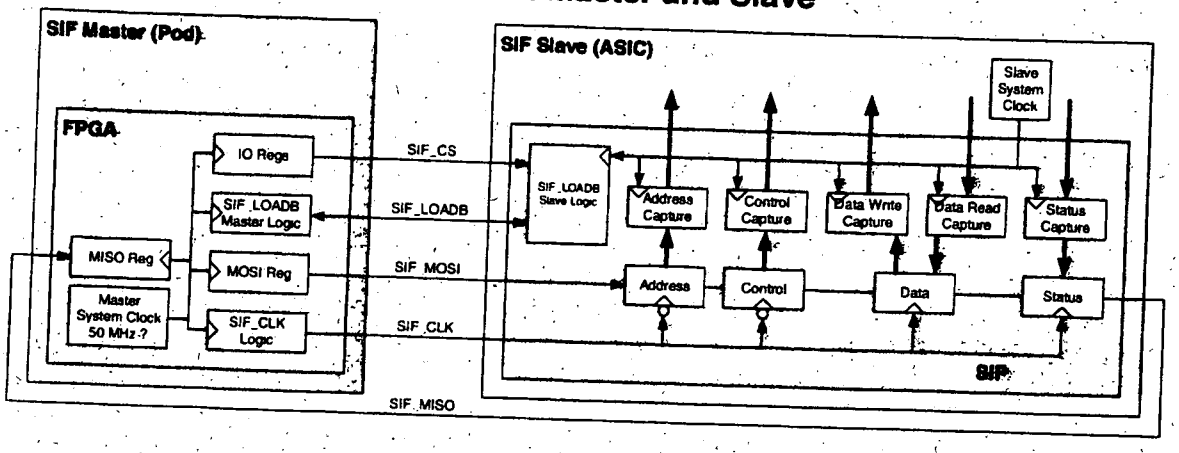
- Pulls SIF_REQ high
- Waits for SIF_ACK to go high
- Enables its SIF signals (SIF_CS, SIF_CLK, SIF_MOSI, SIF_LOADB)
- Performs its SIF Operations

When the SIF Pod wants to relinquish control of the SIF it :

- Tri-states its SIF signals (SIF_CS, SIF_CLK, SIF_MOSI, SIF_LOADB)
- Pulls SIF_REQ low
- Waits for SIF_ACK to go low

If there is more than one SIF Master on the SIF Slave PCB, then they should be daisy chained using extra SIF_REQ and SIF_ACK signals as shown in the diagram above.

## 6.4 Shift Registers and Clocks in Master and Slave

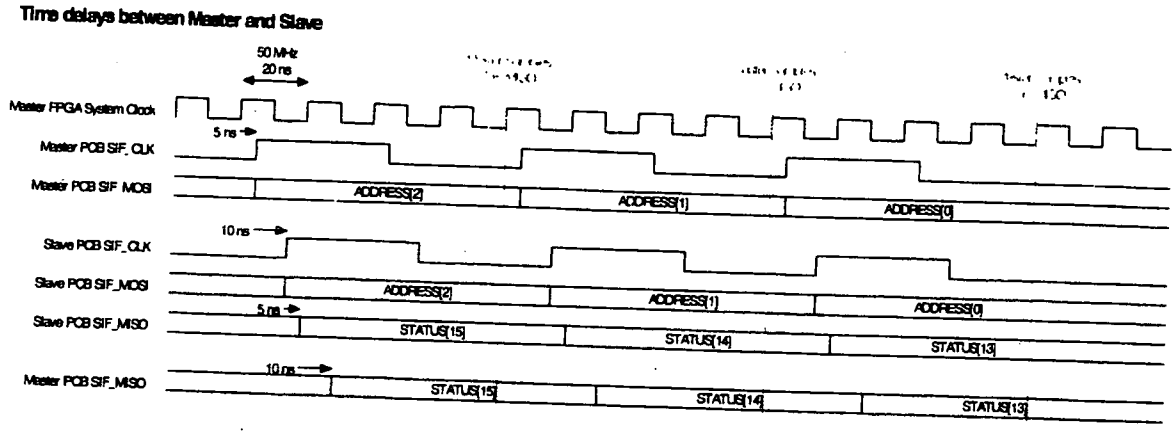



The above diagram shows how information is sent from the Master to the Slave and how it receives information back from the Slave on SIF_MISO. The whole process is controlled by the Master. The SIF Operations are never initiated by the SIF Slave. The SIF Slave only responds to requests from the SIF Master.

This diagram also shows how clocks are used in the Master and Slave :

- SIF_CLK is generated by the Master. It is not used as a clock inside the Master.
- SIF_CLK is used as a clock by the 4 shift register fields in the Slave; Address and Control on negedge, Data and Status on posedge.
- Everything in the Master is clocked by its System clock. This will be a fast continuous running clock (probably 50 MHz).
- SIF_MOSI and SIF_CLK change on posedge MasterSystemClk. SIF_MOSI changes at the point where a posedge is generated on SIF_CLK.
- SIF_MISO is sampled into the Master on posedge MasterSystemClk. The sample can occur at any time on or after negedge SIF_CLK, but must be before posedge SIF_CLK. The fastest shift frequencies will be achieved if this sample point is as late as possible. In practise this means that the sample should be on the last posedge MasterSystemClk before posedge SIF_CLK.

## 6.5 Shift Register Timing Delays between Master and Slave




The above timing diagram shows a SIF Operation between the Master and Slave where :

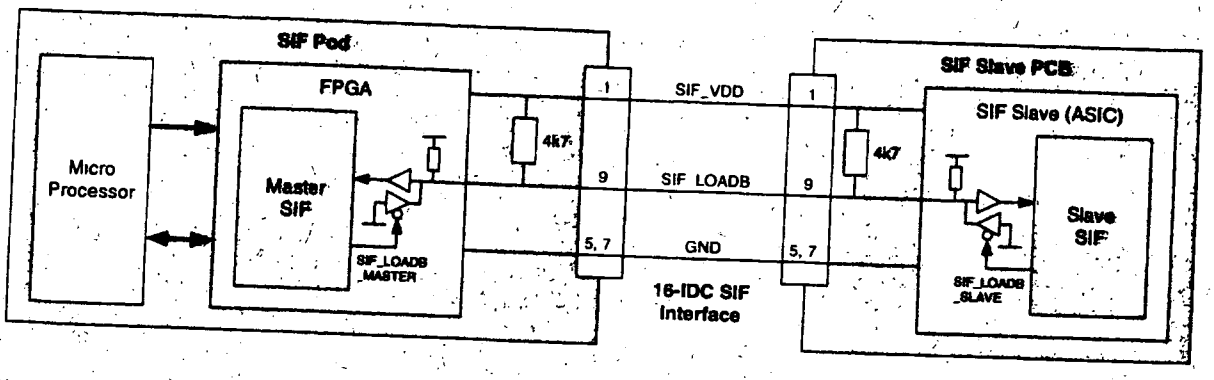
- Only 3 bits are shifted. ADDRESS[2:0] from Master to Slave on SIF_MOSI. STATUS[15:13] from Slave to Master on SIF_MISO.
- Master System Clock = 50 MHz (period = 20 ns). This means that the fastest SIF_CLK available is 25 MHz.
- SIF_CLK = 12.5 MHz (period = 80 ns).
- Silicon delay from Master System Clock to SIF_MOSI and SIF_CLK = 5ns.
- Cable delay from Master to Slave on SIF_MOSI and SIF_CLK = 10 ns.
- Silicon delay from SIF_CLK to SIF_MISO in Slave = 5 ns.
- Cable delay from Slave to Master on SIF_MISO = 10 ns.

The delay times are estimated and will vary from system to system. This shows the best time for the Master to sample SIF_MISO in red.

The Master and Slave can be thought of as one continuous shift register. However it is not symmetrical as SIF_CLK is generated in the Master. Shift registers go wrong if the clock to a downstream bit is late. They do not go wrong if the clock to a downstream bit is early. This is why it is safe to sample SIF_MISO in the Master almost on posedge SIF_CLK. It would not be safe to sample SIF_MOSI in the Slave on posedge SIF_CLK, which is why it is actually sampled on negedge SIF_CLK. So the summary operation of the Master and Slave shift Registers are :

- Master output SIF_MOSI changes on posedge SIF_CLK (though SIF_MOSI is not clocked by SIF_CLK).
- Slave output SIF_MISO changes on posedge SIF_CLK.
- SIF_MOSI is sampled into Slave on negedge SIF_CLK.
- SIF_MISO is sampled into Master just before posedge SIF_CLK.

## 6.6 Master-Slave Handshake on SIF_LOADB




SIF_LOADB is the bidirectional handshaking signal between the SIF Master and the SIF Slaves. The SIF_LOADB pins in Master and Slaves are open-drain bidirectionals with pullup resistors. In addition there is a pullup resistor (normally 10k ohm) on the wire between the Master and Slaves. SIF_LOADB can be pulled low by the SIF Master or any SIF Slave.

SIF_LOADB is active-low. It is high when the SIF is not being used or when the SIF Master is shifting data to and from the SIF Slave. When SIF_LOADB is high, the Master can shift data through the Slave's shift register as much as it likes (taking care not to invoke Command mode !). Nothing will happen in the SIF Slave until the SIF Master pulls SIF_LOADB low.

A SIF Instruction (Read or Write) consists of 3 phases :

- SIF Master shifts the SIF request into the {Address, Control, Data} fields of the SIF Slave shift register.
- Handshake between SIF Master and SIF Slave using the SIF_LOADB signal.
- SIF Master shifts the SIF result out of the {Data, Status} fields of the SIF Slave shift register.

During the handshake phase, the SIF Master must :

- Hold SIF_CS high
- Hold SIF_CLK low
- Not change the value on SIF_MOSI

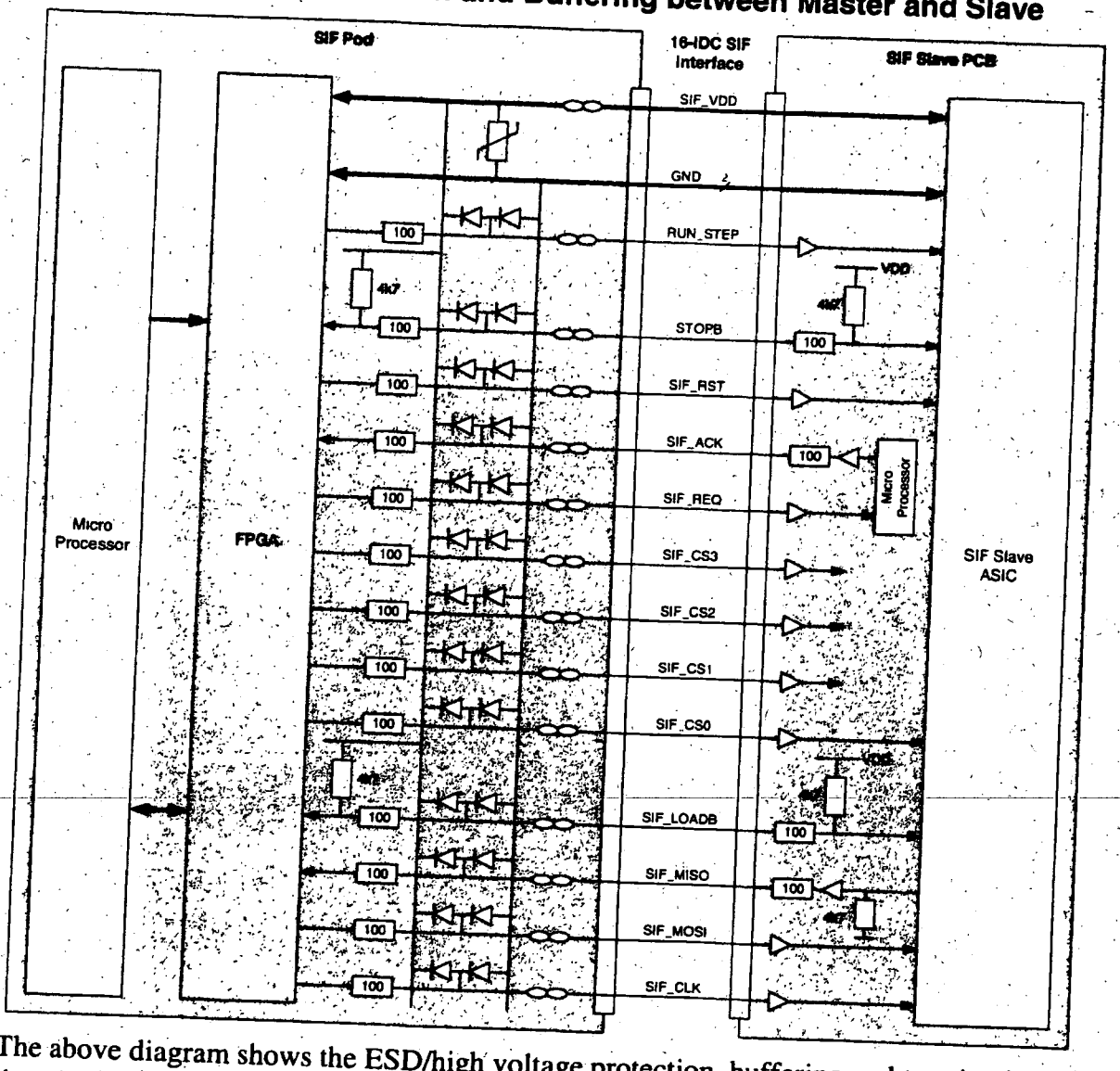
The handshake phase consists of :

- SIF Master pulls SIF_LOADB_MASTER low to indicate that the SIF shift register fields are correctly formatted and that the SIF Slave should perform the requested SIF Read or SIF Write. The Master must hold SIF_LOADB_MASTER low for at least 3 CLK cycles (CLK = SIF Slave system clock). It is safe to assume that CLK frequency is always > 1 MHz. Therefore 3us is the longest SIF_LOADB pulse time ever needed.

- SIF Master stops pulling SIF_LOADB_MASTER low. SIF_LOADB will remain low because of the SIF Slave.
- SIF Slave holds SIF_LOADB low until it has completed the requested SIF Read or SIF Write. If the selected SIF Processor is running, this will not be until a SIF or SLEEPSIF instruction is executed. When the SIF Slave stops pulling SIF_LOADB low, the signal will return high (because of the pullup resistor on SIF_LOADB).
- When the SIF Master sees SIF_LOADB go high, it knows that the requested SIF Read or Write has been completed. This means that it is able to proceed to the next shift phase.

If the SIF Master continues to pull SIF_LOADB low after the SIF Slave has let go, the SIF Slave will not perform any further SIF Instructions. The SIF Slave cannot perform another SIF Instruction until it sees SIF_LOADB go high and then low again.

## 6.7 Protection, Termination and Buffering between Master and Slave




The above diagram shows the ESD/high voltage protection, buffering and termination that should be used in the Pod and Slave. The diagram shows all 13 SIF signals being used in the Slave PCB. In most cases it will only be the bottom 4 or 5 SIF signals that are actually used :

- SIF_CS is not always implemented in the SIF Slave.
- SIF_REQ and SIF_ACK are only used for Multi-Master systems (which is rare).
- SIF_RST is not normally needed (reset instructions can be sent over the 4 main SIF signals).
- STOPB and RUN_STEP are only needed for XAP1. They are not used by XAP2 or XAP3.

It was mentioned earlier that the circuitry required in the Pod is a compromise between the ESD/high voltage requirements and the termination requirements. The above scheme shows how the 13 signal pins are protected by a 100 ohm resistor, diodes (to SIF_VDD and GND) and a polyswitch fuse. Component numbers and values are given in section 9 on the SIF Master.

The Slave PCB is shown with buffers for all signals. These are recommended for FTB (Functional Test Board) and ATB (Analogue Test Board). They are not mandatory for production boards. The benefit of the buffers is :

- Good drive capability.
- Protects ASICs from EMC on the cable.
- Known speeds over various lengths of cable .
- Maintain point-to-point cable connections even for multiple SIF Slaves.

The preferred buffers are NC7NZ17 in the US8 package. This is Fairchild TinyLogic. Each 8-pin pack contains 3 non-inverting buffers. They can deliver 24mA and have a propagation time of 3.6ns (typical) into a 50pF load. They are rated for operation over –40 to +85 degC. There may be some cases where an ATB needs to use a different buffer if the environmental test chamber needs to go above +85 degC (sometimes we need +125 degC). These devices are very fast and therefore must have good de-coupling capacitors between SIF_VDD and GND (e.g 100nF, 0603, X7R). Without this there will be interference on SIF_VDD when the buffers switch.

A 4k7 pulldown resistor is shown at the input to the SIF_MISO buffer. This is needed if the SIF Slave ASIC has a SIF_CS input pin, which tri-states the SIF_MISO ASIC output. The resistor prevents the buffer input from floating to mid-rail.

# 7 SIF Slave

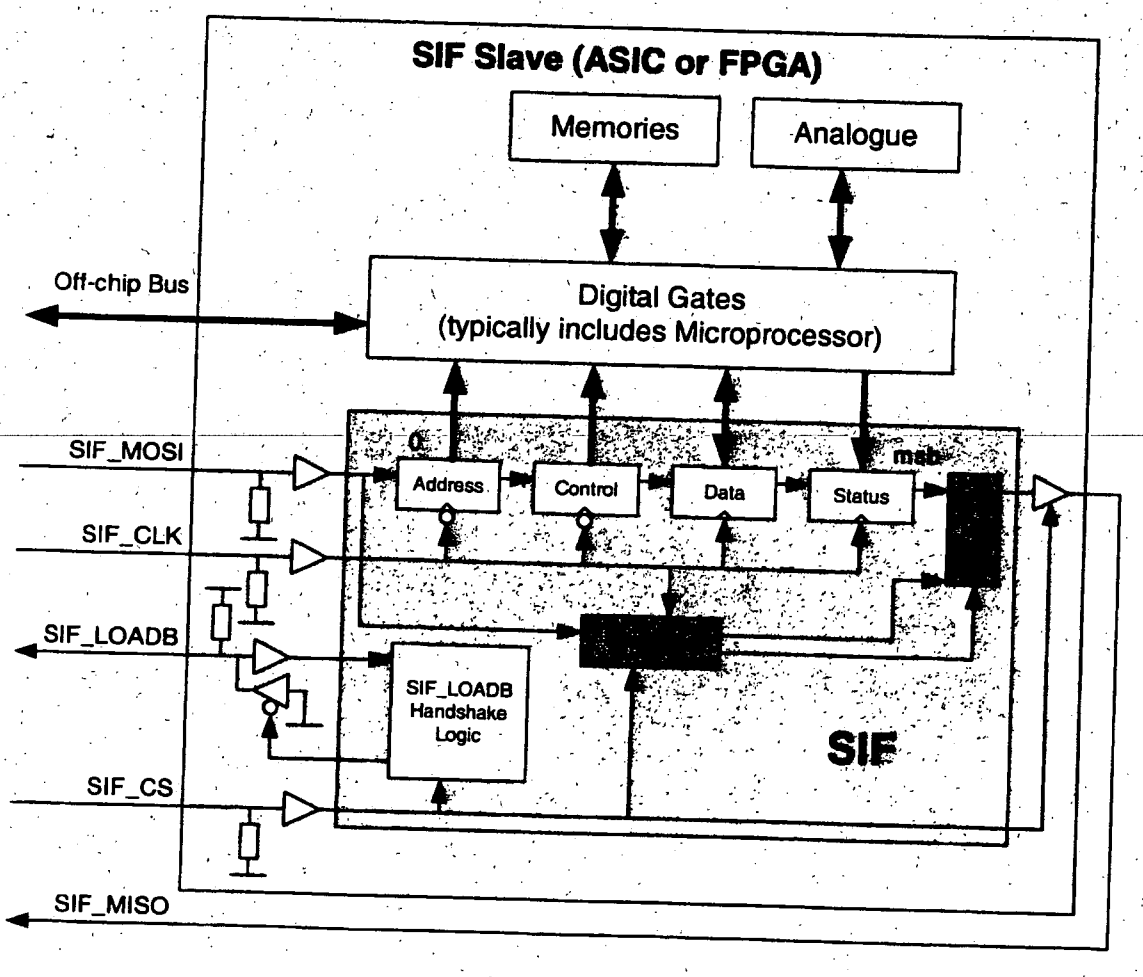
A SIF slave is normally implemented as an ASIC or FPGA.

SIF slaves can operate in Normal or Command mode. All SIFs support Normal mode. The only design to support Command mode as of January 2005 is the XAP3.

The shift register length for Normal mode can vary from one slave design to another.

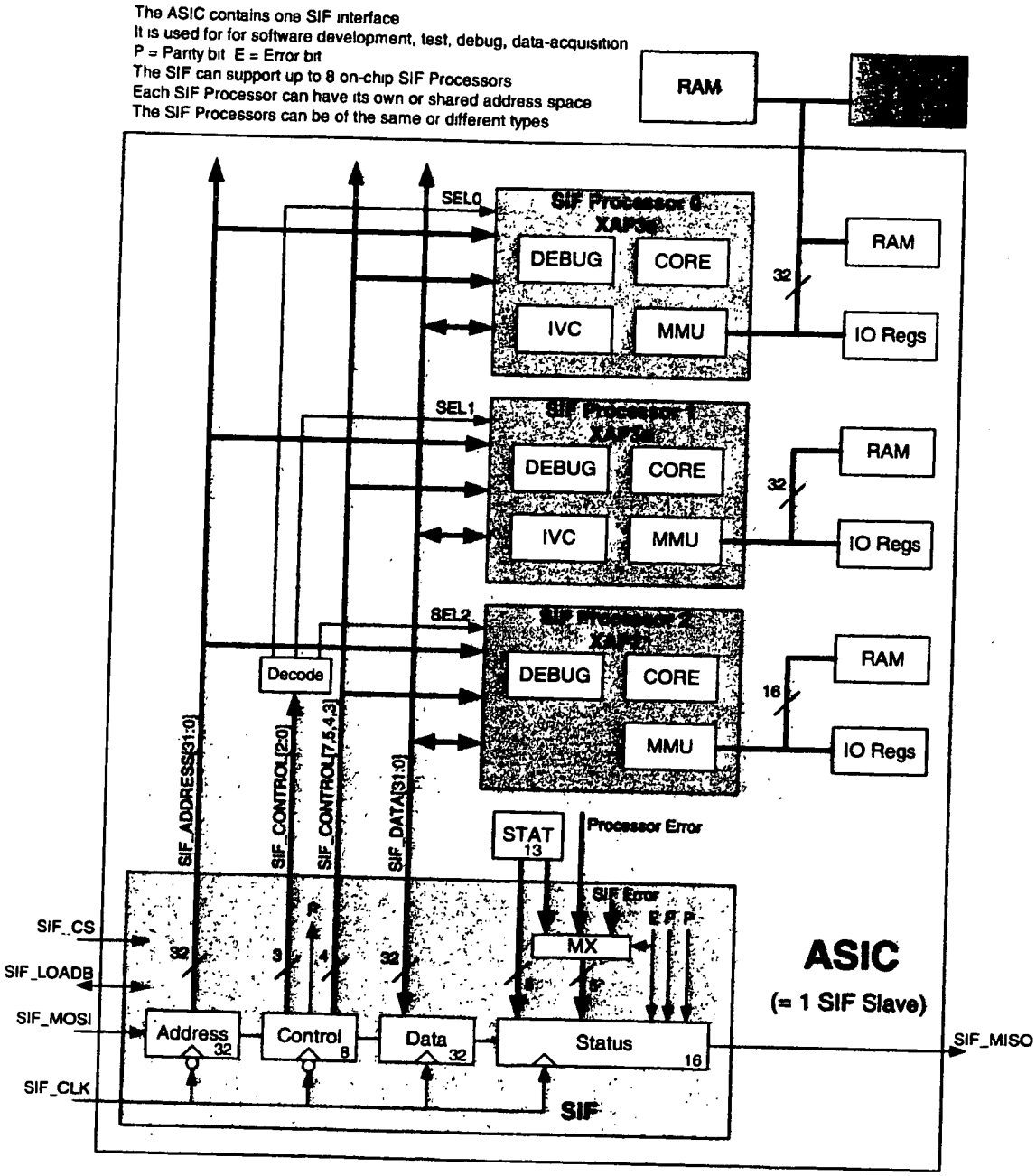
The shift register length for Command mode is always 8 bits. Command mode operates identically across all SIF slaves. It enables the SIF Master to discover the SIF configuration details of the SIF slaves in Normal mode (e.g shift register field lengths).

## 7.1 Block Diagram




The SIF is in the yellow block. The white blocks within it are for Normal mode. The purple blocks within it are for Command mode (and don't exist in older SIF slaves).

## 7.2 SIF Slave containing Multiple SIF Processors




A SIF Slave should only ever contain 1 SIF interface. However, this can support several processors, as shown in the above diagram. These processors can be of different types. The default SIF supplied with the XAP3 supports up to 8 on-chip SIF Processors.

## 7.3 SIF Pins

More information on the SIF pins and pad types is given in section 3 on the 16-IDC SIF Interface.

An ASIC (SIF Slave) containing the SIF will have the following pins :

| *SIF Signal* | *Driven By* | *Description* |
|---|---|---|
| SIF_CS | Master | Chip Select (active high) |
| SIF_LOADB | Master & Slave(s) | Handshake signal (active low) |
| SIF_CLK | Master | Clock |
| SIF_MOSI | Master | Master Out, Slave In (data) |
| SIF_MISO | Slave(s) | Master In, Slave Out (data) |

### *7.3.1 SIF Slave Pin Order for Package and Silicon Die*

Most ASIC packages have pins identified by a single number (QFP, QFN, PLCC, TSOP, SOIC, DIL, etc). The SIF signals should be allocated to package pins in the following order :

| *SIF Signal* | *Package pin number* | *Silicon die pad number* |
|---|---|---|
| SIF_CS | M | N |
| SIF_MISO | M + 1 | N + 1 |
| SIF_MOSI | M + 2 | N + 2 |
| SIF_CLK | M + 3 | N + 3 |
| SIF_LOADB | M + 4 | N + 4 |

Similarly the SIF pads should all be contiguous and in this order on the silicon die. The pin and pad numbers are expected to increase in an anti-clockwise direction when looking down on the package or silicon die.

BGA packages have pins in a 2D grid of rows and columns. Consequently they use pin names of the form B14. The SIF Pads should be allocated to the BGA pins in a sensible manner, given the above constraint on pad positions on the silicon die.

### *7.3.2 Sharing SIF Pins with JTAG*

3 of the SIF pins can be shared with JTAG pins as follows :

| *SIF Pin* | *JTAG Pin* | *Comment* |
|---|---|---|
| SIF_CS | | |
| SIF_LOADB | | |
| SIF_CLK | TCK | |
| SIF_MOSI | TDI | Sampled on posedge TCK |
| SIF_MISO | TDO | Changes on negedge TCK |
| | TMS | extra pin |

Separate JTAG controller logic should be included inside the ASIC. This will use the clock edges as defined above (which is the opposite from normal SIF operation) :

- JTAG samples input on posedge TCK, changes output on negedge TCK
- SIF samples input on negedge SIF_CLK, changes output on posedge SIF_CLK.

SIF leaves SIF_CLK low when not being used. This means that :

- First edge (posedge) changes the Master and Slave outputs
- Second edge (negedge) samples the Master and Slave inputs.

JTAG can use a free running TCK (operation controlled by TMS) or a gated TCK. In the gated case, TCK is low when not used. This is different from SIF and means that :

- First edge (posedge) samples the Master and Slave inputs
- Second edge (negedge) changes the Master and Slave outputs.

Most JTAG interfaces use a 14-IDC or 20-IDC connector. This often contains TCK_RET. This is a returned clock from the Slave to the Master. TCK is synchronised with the Slave system clock. The Master operates a hand-shaking system called Adpative Clocking where it won't generate posedge TCK until it receives negedge TCK_RET. Similarly the Master won't generate negedge TCK until it receives posedge TCK_RET. This enables the Slave to slow down the Master, which is sometimes necessary if theSlave is in a slow low-power mode. However it does reduce the maximum shift rate for the JTAG interface.

None of this is necessary in SIF because the SIF shift register is completely asynchronous from the Slave's system clock. The handshaking between Master and Slave is handled by the SIF_LOADB signal. This means that even when the Slave is in a slow low power mode, the Master can still use the fastest shift rates to communicate with the Slave.

### *7.3.3 Sharing SIF Pins with Scan Path*

3 of the SIF pins can be shared with Scan Path pins as follows :

| *SIF Pin* | *Scanpath Pin* | *Comment* |
|---|---|---|
| SIF_CS | | |
| SIF_LOADB | | |
| SIF_CLK | scan_clk | |
| SIF_MOSI | scan_in | Sampled on posedge scan_clk |
| SIF_MISO | scan_out | Changes on negedge scan_clk |

## 7.4 Shift Register Fields

The SIF shift register consists of 4 fields. The length of these fields can each be from 0 to 32 bits. This is a property of the slave design & remains fixed. The Computer and SIF Master must know the lengths of these 4 fields in order to communicate with the SIF

Slave. This is either known by manual configuration or (on newer designs) by auto-discovery using SIF Commands.

For all 4 fields, data is shifted in and out MSB first. The order of the 4 shift register fields in the SIF Slave from input to output must always be :

- SIF_MOSI pin
- Address field
- Control field
- Data field
- Status field
- SIF_MISO pin

### 7.4.1 Address Field

The Address field is an output from the SIF Master and an input to the SIF Slave. Address steps are in the same units as those used by the selected SIF Processor in the SIF slave. e.g :

- XAP1 and XAP2 addresses represent 16-bit steps.
- XAP3 address represents 8-bit steps (i.e byte-addressing).

### 7.4.2 Control Field

The Control field is an output from the SIF Master and an input to the SIF Slave. It can include several different signals, which from LSB to MSB must be in the following order :

| *Position* | *Bits* | *XAP usage* | *Comment* |
|---|---|---|---|
| LSB | PROG_DATAB | XAP1, XAP2 | Specifies Program or Data space for Harvard architecture processors. |
| | PROCESSOR | XAP3 | Selects processor/address space. Used for multiple processor/address spaces within a single SIF slave. XAP3 has 3 bits to support up to 8 processors in the same SIF slave. |
| | DEBUG | XAP2, XAP3 | 0 = Normal SIF Instruction<br>1 = Debug SIF Instruction |
| | SIZE | XAP2, XAP3 | Specifies size of Data word for Read or Write. |
| | PARITY_AC | XAP3 | Odd Parity (odd number of 1s) on |

| | | | |
|---|---|---|---|
| | | | Address and Control fields (including the parity bit itself). If used, must be MSB-1 of the Control field. |
| MSB | WRITE | XAP1, XAP2, XAP3 | Write must always be the MSB of the Control field. Can be active high or active low (depends on slave design). |

The length of the Control field is defined to include the PARITY_AC and WRITE bits. Many functions in the xSIF API pass a Control argument. Such Control arguments should :

- Not include the WRITE bit in SIF Read and Write functions (as the function already implies the required value for the WRITE bit).
- Include the WRITE bit in SIF Cycle functions (as they can be used generically for SIF reads and writes).

Similarly with the PARITY_AC bit, Control arguments in the xSIF API should :

- Not include the PARITY_AC bit in SIF Read and Write functions (as the xSIF driver will automatically generate the PARITY_AC bit).
- Include the PARITY_AC bit in SIF Cycle functions (as they should be able to explicitly set all bits, causing parity errors or not, as desired).

### *7.4.3 Data Field*

The Data field for a SIF Write is an output from the SIF Master and an input to the SIF Slave.

The Data field for a SIF Read is an output from the SIF Slave and an input to the SIF Master.

The memory or register that the data is written to or read from depends on the Address and Control fields used for the SIF Instruction.

The Shift Register Data field should only be updated for valid SIF Reads. It should not be updated for SIF Writes or any SIF Error.

### *7.4.4 Status Field*

The Status field is always an output from the SIF Slave and an input to the SIF Master (regardless of whether it is a SIF read or a write). The Shift Register Status field is updated after all completed (i.e complete SIF_LOADB handshake) SIF Instructions (valid or error).

The value of the Status field is independent of the Address and Control fields. It is normally updated with an internal hardware register, STAT[] and error information on the success of the SIF Instruction.

The SIF Slave hardware module (Verilog) has an input bus for STAT[] that feeds into the Status field. The hardware designer can decide what information should be connected to this bus. Examples include :

- TimeStamp. i.e a counter (say 20 kHz) that will indicate the exact time at which the SIF read or write took place. LSB = Highest frequency bit of Counter.
- Memory-mapped register. SIF Status information can be selected by on-chip software (which can be changed after the silicon is fixed).

The Status field is an output from the SIF Slave and an input to the SIF Master. It can include several different signals, which from LSB to MSB must be in the following order :

| *Position* | *Bits* | *XAP usage* | *Comment* |
|---|---|---|---|
| LSB | STAT | XAP1, XAP2, XAP3 | Hardware register in ASIC.<br>Often used as a TimeStamp (with bit-0 as the highest frequency bit). |
| | ERROR_CODE | XAP3 | Details the SIF or Processor Error |
| | ERROR_TYPE | XAP3 | 0 = SIF Error<br>1 = Selected Processor Error |
| | ERROR | XAP3 | 0 = No error in SIF Instruction<br>1 = SIF Instruction could not be executed as there was a SIF or Processor error. |
| | PARITY_D | XAP3 | Odd Parity (odd number of 1s) on Data field (including the parity bit itself). If used, must be MSB-1 of the Status field. |
| MSB | PARITY_S | XAP3 | Odd Parity (odd number of 1s) on Status field (including the parity bit itself). If used, must be MSB of the Status field. |

If there is no error, the ERROR_TYPE and ERROR_CODE fields can be used for higher bits in STAT[].

PARITY_D and PARITY_S are set by the SIF Slave and checked by the SIF Master. PARITY_D should be set first. PARITY_S is set afterwards and covers the whole Status field (i.e includes PARITY_D).

## 7.5 XAP and APE Processors

The XAP1, XAP2 and XAP3 processors all include a SIF interface. The APE1 and APE2 processing engines normally contain a SIF interface. The standard lengths for these SIF implementations are :

| *Processor* | *Address Bits* | *Control Bits (inc Write Bit)* | *Data Bits* | *Status Bits* | *Total Bits* | *Active Write* |
|---|---|---|---|---|---|---|
| XAP1 | 16 | 2 | 18 | 0 | 36 | 0 |
| XAP2 | 24 | 8 | 32 | 0 | 64 | 0 |
| XAP3 | 32 | 8 | 32 | 16 | 88 | 1 |
| XAP4 | 16 | 8 | 16 | 12 | 52 | 1 |
| XAP5 | 24 | 8 | 16 | 20 | 68 | 1 |
| APE1 | 15 | 1 | 32 | 0 | 48 | 1 |
| APE2 | 15 | 1 | 16 | 0 | 32 | 0 |

## 7.6 Verilog RTL Implementation

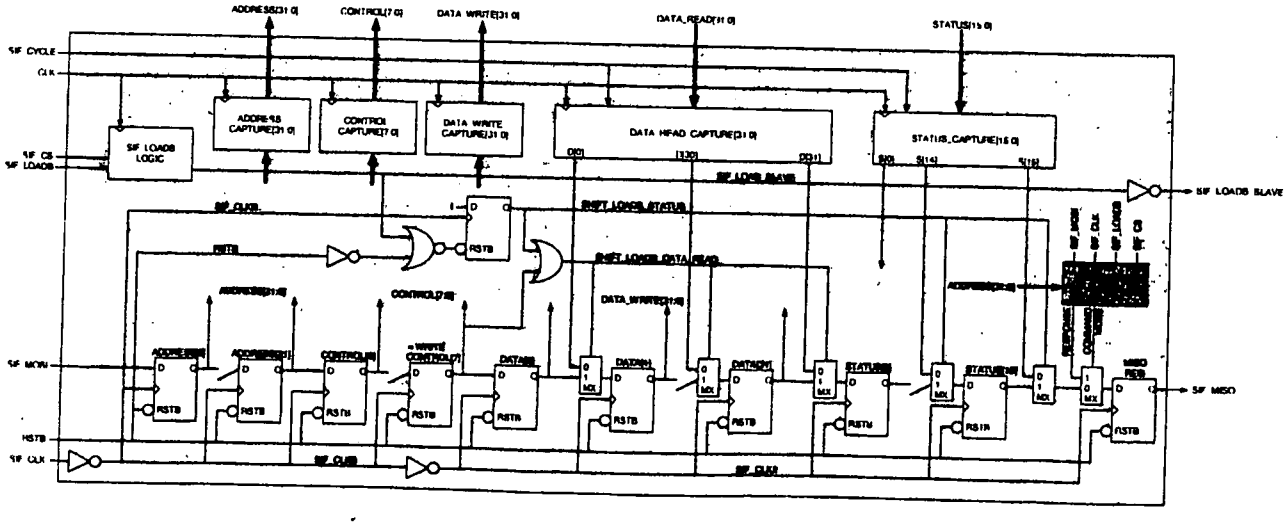
This describes the Verilog xsif module stored in the CCL CVS ASIC Repository.

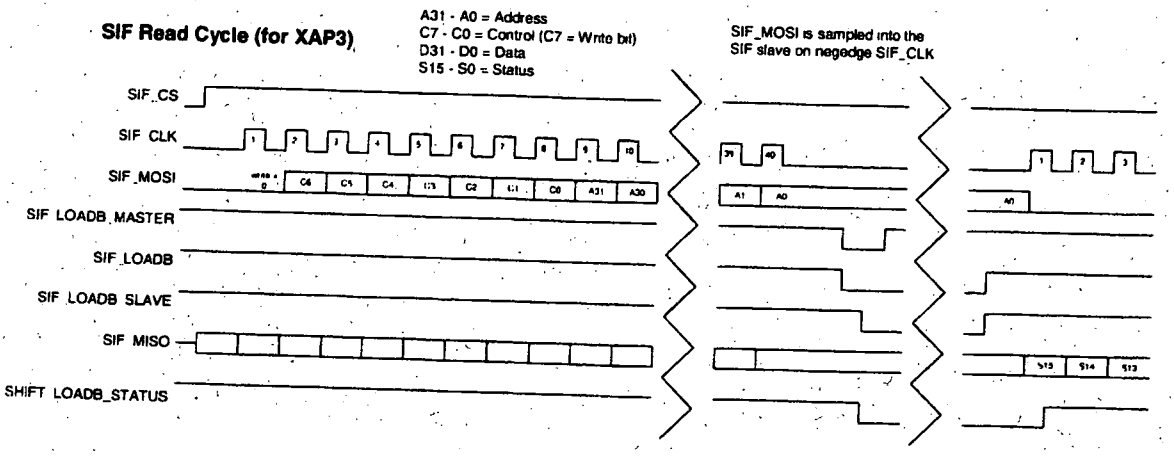
The SIF Slave implementation below is for a XAP3. Of course it can be scaled to have different field lengths (Address, Control, Data, Status) for other SIF Slave devices.

This circuit uses 2 clocks :

- SIF_CLK - used for Shift register and SIF Command logic.
- CLK (= SIF Slave system clock) – used for SIF_LOADB logic and the Capture registers for Address, Control, Data_Write, Data_Read and Status.

This circuitry must work whether SIF_CLK is faster or slower than CLK.






The shift-in and shift-out phases are only clocked by SIF_CLK (do not use CLK at all).

The handshake (SIF_LOADB pulse) phase is only clocked by CLK. Negedge SIF_LOADB is detected synchronously with CLK. Thus SIF_LOADB_SLAVE is clocked by CLK.

After negedge SIF_LOADB has been detected, the Address, Control and Data shift registers are copied to the Address, Control and Data_Write capture registers (which are clocked by CLK). This means that the shift register can be clocked by SIF_CLK afterwards (e.g for a SIF Cancel) without corrupting the Address, Control and Data_Write fields used for the SIF Instruction (in case it is executed before the SIF Cancel has completed).

The Address and Control fields are clocked on negedge SIF_CLK. They always operate as a shift register.

The Data and Status fields and MISO REG flipflop are clocked on posedge SIF_CLK. They normally operate as a shift register. However, they sometimes do a parallel load from the Data_Read and Status Capture registers. These occur on the first posedge SIF_CLK after posedge SIF_LOADB :

- Happens for Data field for valid SIF Reads only (controlled by SHIFT_LOADB_DATA).
- Always happens for Status field (controlled by SHIFT_LOADB_STATUS).

When a parallel load happens, the Capture register bits are shifted up by one bit, when fed into the Data and Status fields. This means that there is never a parallel load into DATA[0]. DATA[0] always acts as a shift register, taking its input from CONTROL[MSB].

The 2 SHIFT_LOADB_* control signals :

- Go low on negedge SIF_LOADB_SLAVE (synchronous to CLK)
- Go high on first negedge SIF_CLK after posedge SIF_LOADB_SLAVE.

# 8 SIF Slave PCB

## 8.1 Strategies

This section contains several diagrams that show the extra components that should be placed on the SIF Slave PCB for the most common configurations of SIF Slave ASICs. These use the 4 main SIF signals (SIF_LOADB, SIF_CLK, SIF_MOSI, SIF_MISO) and one or more SIF_CS signals. The components are for buffering and termination. The diagram in section 6.7 shows the components needed in a Slave PCB if all 13 SIF signals are used (which is unusual).

All of these extra components should be placed right next to the 16-IDC connector on the SIF Slave PCB. The SIF Slave ASICs should also be placed as close as possible to the connector. All of the buffers should be powered by the SIF_VDD and GND signals in the 16-IDC interface and should have 100nF de-coupling capacitors.

The general strategy is that production boards should minimise the number of components fitted. This can be restricted to a few 0603 resistors. This minimises cost but does not provide as much drive strength or protection as a board that contains buffers. It therefore may have to be used at lower shift speeds and because there is less protection should be used with short cables.

Functional Test boards (FTB) and Analogue Test Boards (ATB) are made in lower volumes and can afford to have more components in order to provide :

- Greater shift speeds
- Longer cables
- Greater EMC and ESD protection

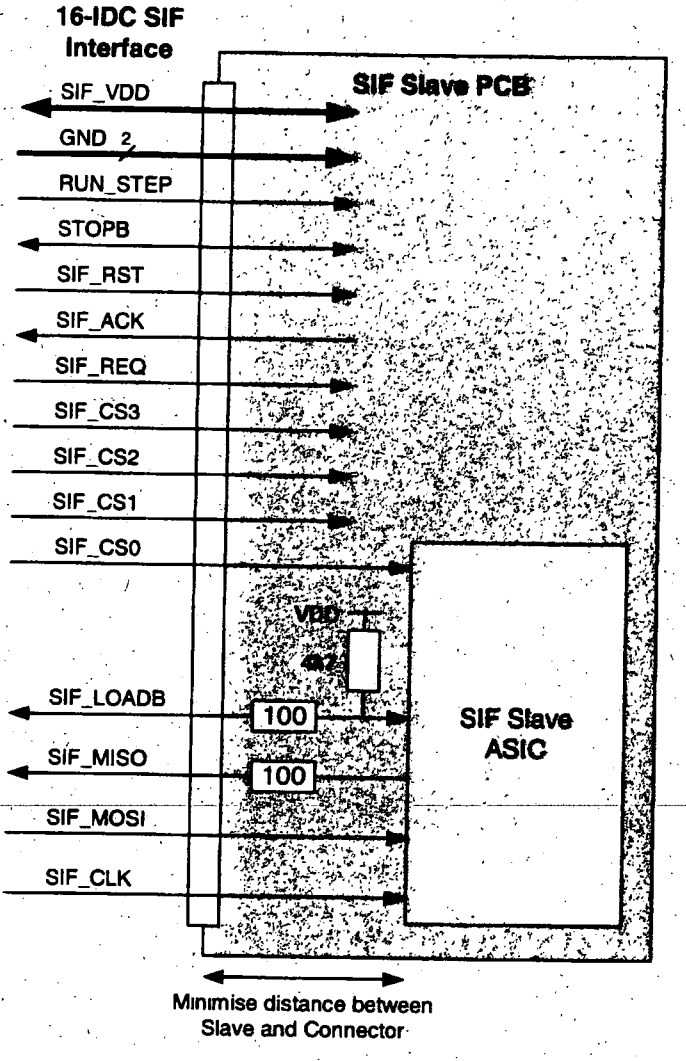
The preferred buffers are NC7NZ17 in the US8 package. This is Fairchild TinyLogic. Each 8-pin pack contains 3 non-inverting buffers. They can deliver 24mA and have a propagation time of 3.6ns (typical) into a 50pF load. They are rated for operation over –40 to +85 degC. There may be some cases where an ATB needs to use a different buffer if the environmental test chamber needs to go above +85 degC (sometimes we need +125 degC). These devices are very fast and therefore must have good de-coupling capacitors between SIF_VDD and GND (e.g 100nF, 0603, X7R). Without this there will be interference on SIF_VDD when the buffers switch.

In multi-slave systems, the buffers are connected in parallel to maintain simple point-to-point links. This reduces reflections and enables faster shift speeds to be used.

The most important signal is SIF_CLK. It is important that there are no oscillations which might cause multiple clock edges at the SIF Slave inputs.

## 8.2 Single Slave

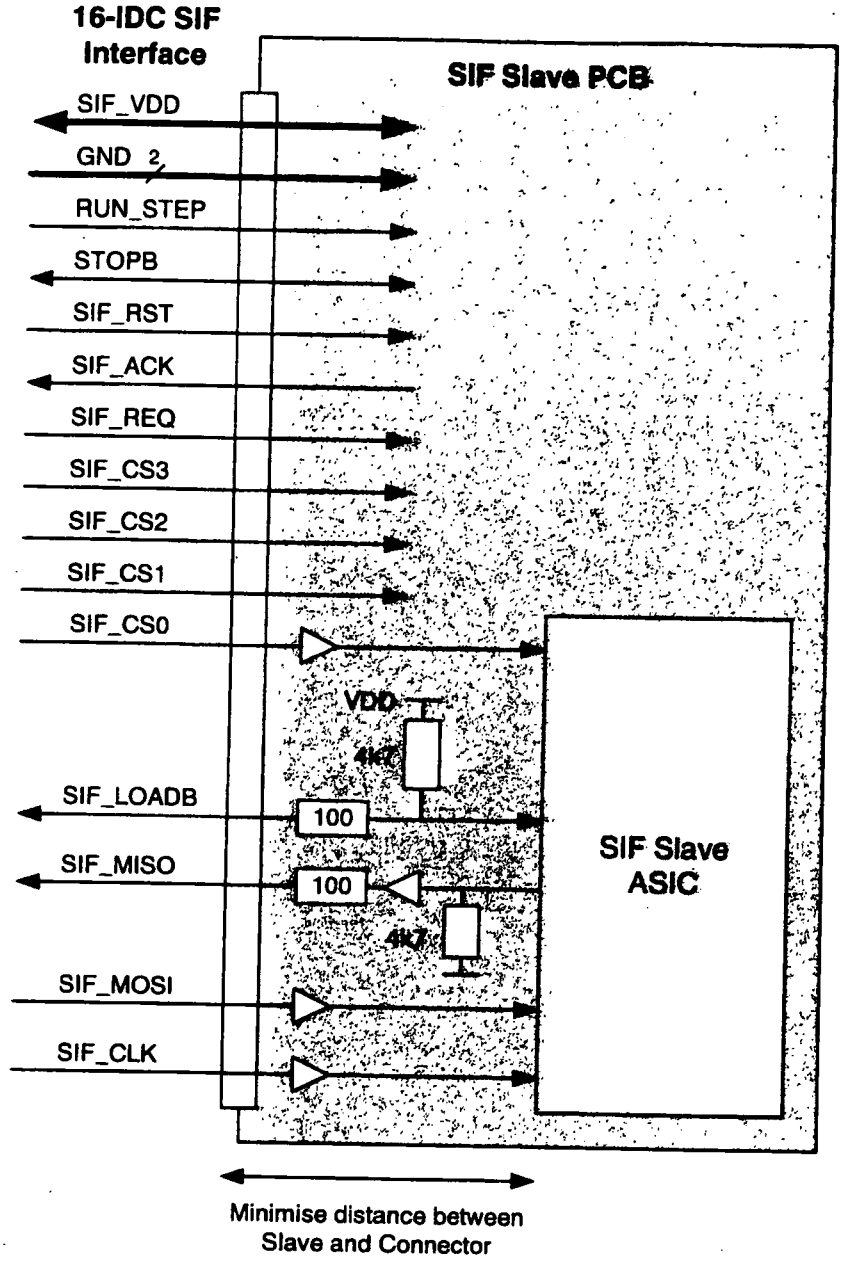
### *8.2.1 Low-cost Production Board*




The SIF_CS pin is not essential for a single-slave system and may not be included in the SIF Slave ASIC.

This board should be used with a short ribbon cable. If you need the board to work with a long cable it would be sensible to add a buffer and pulldown resistor to the SIF_MISO signal (as shown in the following Test version of a single-slave board).

### *8.2.2 Test Board*



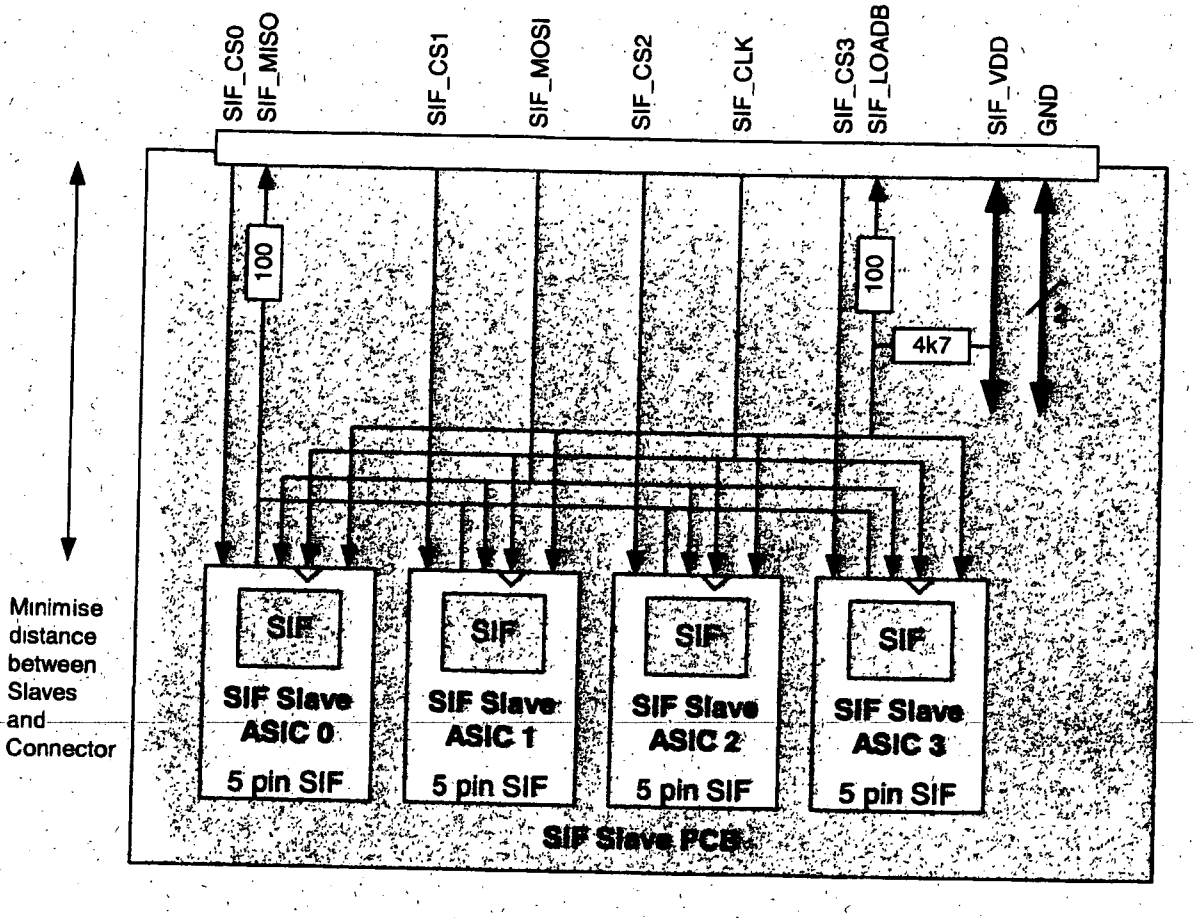

The addition of buffers makes this board faster and more immune to interference than the previous low cost Production board for a Single Slave SIF system.

The SIF_CS pin is not essential for a single-slave system and may not be included in the SIF Slave ASIC. If it is not included then the above components can be omitted :

- Buffer on SIF_CS0.
- 4k7 pulldown resistor on SIF_MISO (because ASIC output will always be enabled).

## 8.3 Multi Slave

### *8.3.1 Low-cost Production Board*



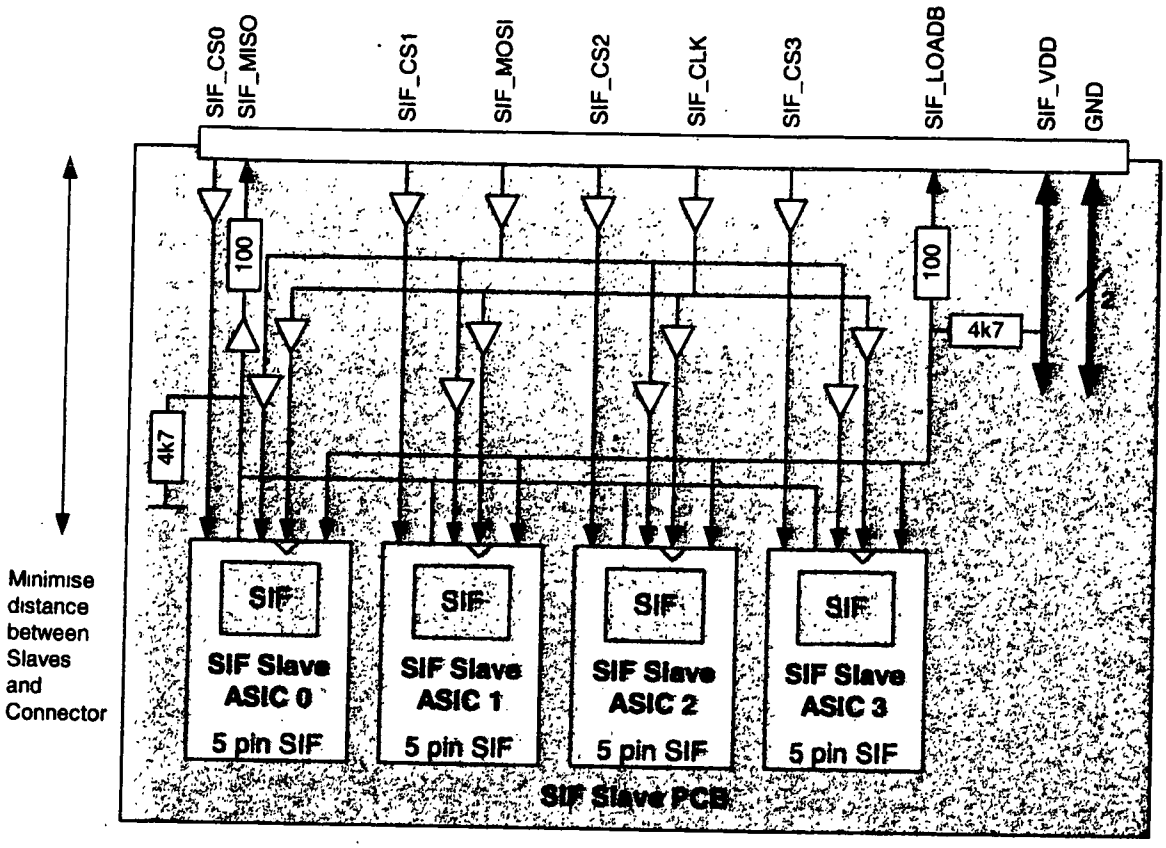

This board should be used with a short ribbon cable. If you need the board to work with a long cable it would be sensible to add a buffer and pulldown resistor to the SIF_MISO signal (as shown in the following Test version of a multi-slave board).

It may need to be used with a modified SIF Pod that generates signals with slow edges so as to avoid reflections. This is especially important on SIF_CLK.

The above scheme can be used for Functional (FTB) or Analogue (ATB) Test Boards.

Of course a similar scheme can be used for a board containing 2 or 3 SIF Slave ASICs.

### *8.3.2 Test Board*




The addition of buffers makes this board faster and more immune to interference than the previous low cost Production board for a Multi Slave SIF system.

The above scheme can be used for Functional (FTB) or Analogue (ATB) Test Boards. It can be used with high speeds and a long ribbon cable to the SIF Pod. Of course a similar scheme can be used for a board containing 2 or 3 SIF Slave ASICs.

All of the external components should be placed close to the SIF Connector.

The SIF Slaves should also be placed as close as possible to the SIF Connector.

The buffers come in packs of 3. Multiple buffers are needed for the SIF_CLK and SIF_MOSI signals. Where possible, all 3 buffers in a pack should be used for the same signal.

# 9 SIF Master

The most commonly used SIF Master is a SIF Pod. Separate microprocessors or ASIC test machines can also be SIF Masters. The implementation example given here is for a SIF Pod.

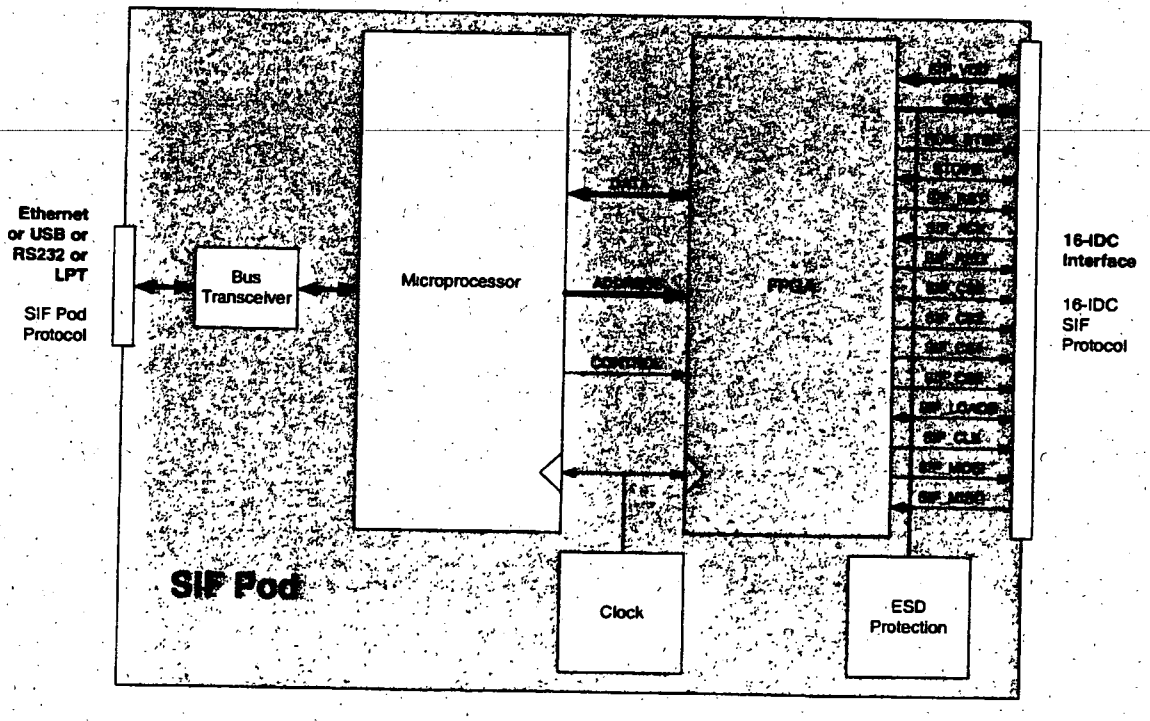
## 9.1 Unique SIF Pod Identifiers

All modern SIF Pods have unique Pod identifiers. These are 16-bit ? in the following ranges :

| *SIF Pod* | *Pod Identifier range* |
|---|---|
| USB SIF Pod | 0x0000 – 0x0FFF |
| Ethernet SIF Pod | 0x1000 – 0x1FFF |
| Universal SIF Pod | 0x2000 – 0x2FFF |

These numbers can be read by the Computer, using the xSIF API.

## 9.2 Block Diagram



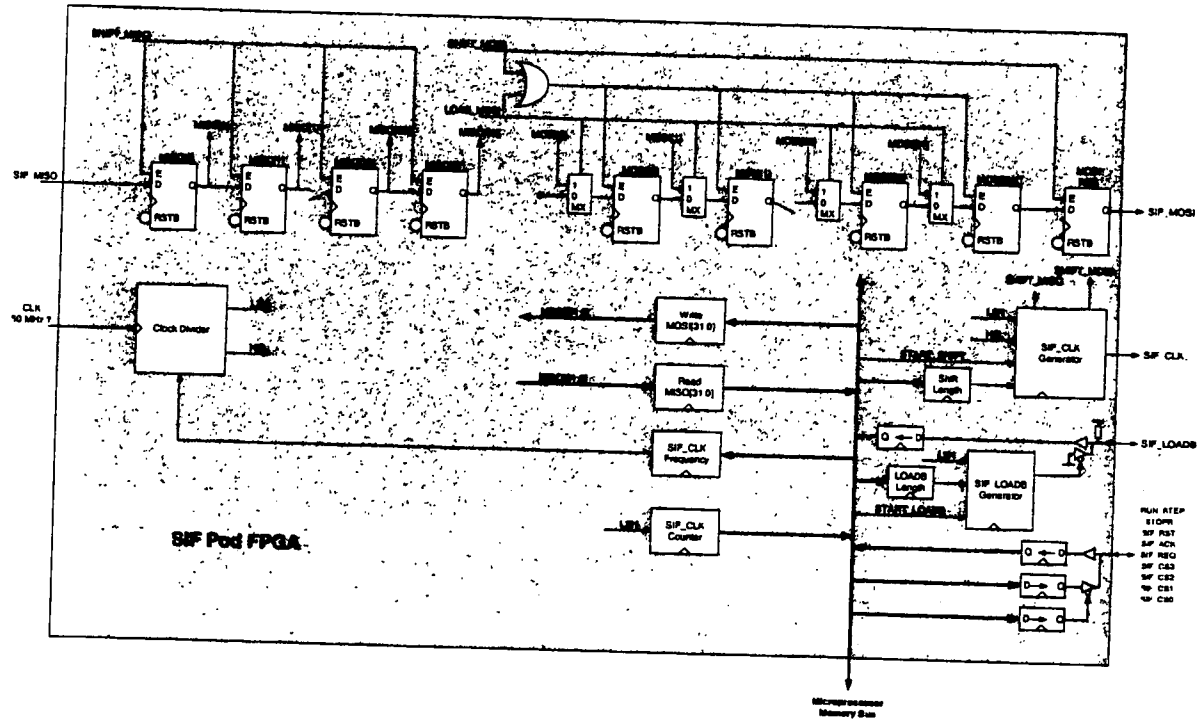

A SIF Pod can be implemented in many different ways. The most common way in modern pods is to use :

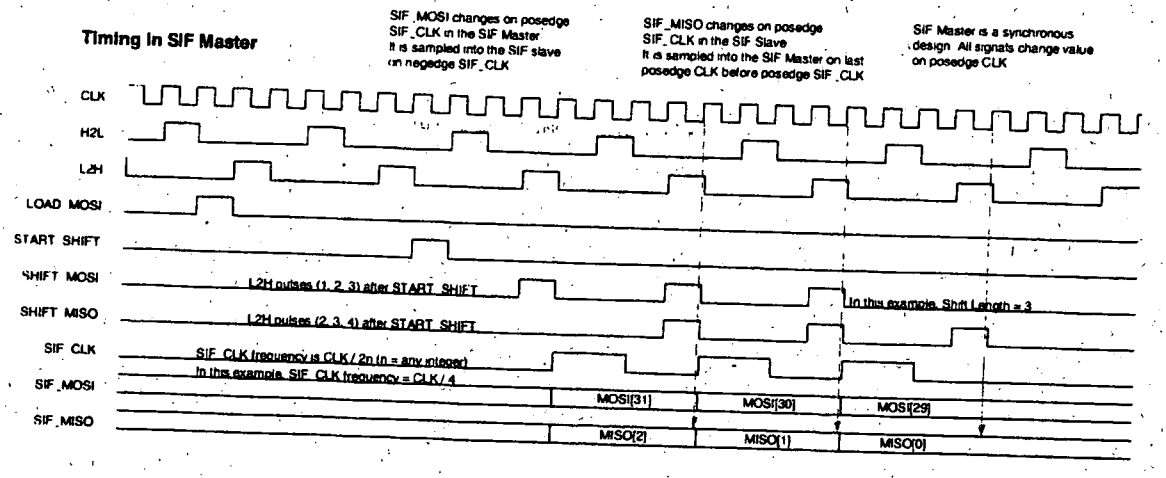
- A microprocessor that includes the MAC (and maybe even the PHY) for the chosen bus interface (e.g USB, Ethernet, RS232) to the computer. The microprocessor must interface with the computer using the SIF Pod Protocol.
- An FPGA (e.g Xilinx, Altera) for implementing the fast logic to the 16-IDC SIF Interface. This is able to implement a controlled timing scheme for the signals. The FPGAs used as of Feb 2005 are Xilinx Virtex2 or Spartan XL. They have good controls for setting pin voltages and impedances. The Spartan XL can supply 24mA at 3V3.
- ESD protection for the various signals in the 16-IDC SIF Interface.

A trade-off can be made between the microprocessor and FPGA, as to which should implement the various parts of the SIF Slave Protocol :

- Simpler to implement if the microprocessor looks after more of the protocol.
- Faster and more controlled signal timings if the FPGA looks after more of the protocol.

## 9.3 Verilog RTL Implementation for FPGA






All the flipflops in this design are clocked on posedge of the same system clock called CLK. This is likely to be a fast continuous clock (e.g 50 MHz). The logic is controlled with enable signals. SIF_CLK is not used as a clock inside the Pod FPGA. It is a generated signal from posedge CLK.

The above FPGA design is a simple one which relies on the external microprocessor to do the control for multi-cycle functions such as xsif_read_array() or xsif_contread_start(). It contains :

- 32-bit MISO register. This samples the SIF_MISO input and shifts on the last posedge CLK before posedge SIF_CLK. The register can be read by the microprocessor bus.
- 32-bit MOSI register. This can be written by the microprocessor bus. It shifts data out and changes the SIF_MOSI output, coincident with posedge SIF_CLK.
- A clock divider that generates the continuous running L2H and H2L control signals. These signals have a pulse width of one CLK period. They precede the posedge and negedge respectively of SIF_CLK. The frequency is set by the SIF_CLK Frequency register.
- SIF_CLK generator. This generates a stream on N pulses on SIF_CLK at the requested frequency. N is specified in the Shift Length register. In the diagram above, N = 3.
- SIF_LOADB generator. Generates the SIF_LOADB_MASTER pulse low of the specified time (set by LOADB Length register).
- 9 IO Registers. Set the direction and output values. Read the input values.
- 4 IO registers to read (but not write) the present values on the 4 main SIF signals (SIF_LOADB, SIF_CLK, SIF_MOSI, SIF_MISO).
- Microprocessor Memory bus interface. External 16 or 32-bit bus interface controls reads and writes to all the above memory-mapped registers (synchronised to CLK).

The MOSI and MISO shift registers could be 64-bit. This would be a bit faster, enabling longer shifts before having to do a parallel load.

### *9.3.1 STOPB and RUN_STEP circuitry*

STOPB and RUN_STEP are debug control signals used for the XAP1 Processor. They are not used for any other processors. These signals control :

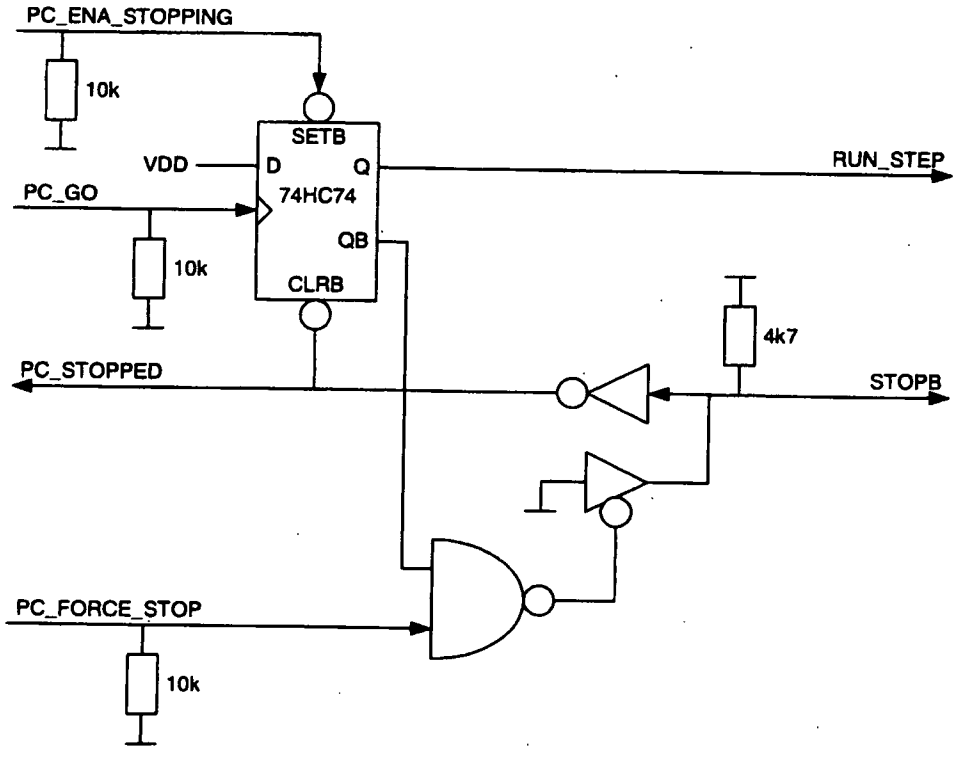
- Start
- Stop
- Single-step
- Run to Breakpoint

In the XAP2 and XAP3 ASICs these functions are implemented as SIF Instructions, which is why they do not have STOPB or RUN_STEP pins.

The LPT SIF Pod does not implement any circuitry for STOPB or RUN_STEP. Pins 15 and 16 in the 16-IDC Interface are left unconnected.

The modern Pods (USB, Ethernet, Universal) all implement the STOPB and RUN_STEP functionality in the FPGA. This is implemented as hardware circuitry the same as in the ISA SIF Pod 98.This is interfaced to the memory bus with the following control signals :

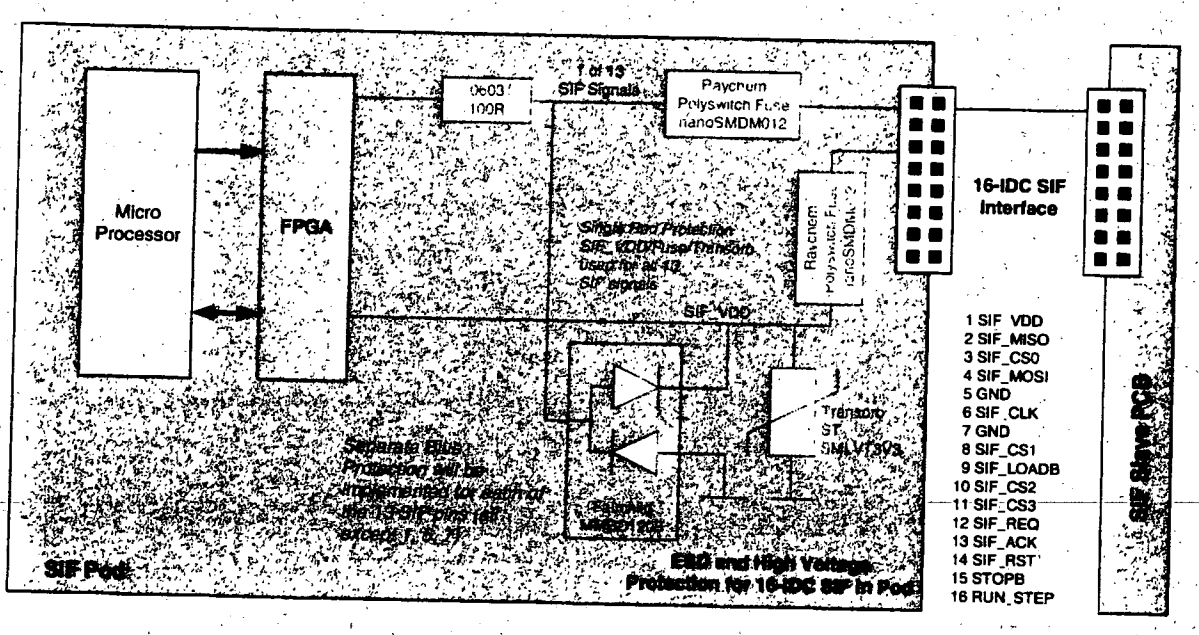
- PC_STOPPED (Pod input)
- PC_ENA_STOPPING (Pod Output)
- PC_GO (Pod Output)
- PC_FORCE_STOP (Pod Output)



## 9.4 ESD and High Voltage Protection in SIF Pod

The recommended circuitry for ESD and high-voltage protection to the 16-IDC interface in a SIF Pod is as follows :

- The red circuit protects SIF_VDD.
- The blue circuits (instanced 13 times) protect the 4 SIF and 9 IO signals.




The 100 ohm series resistors in this protection scheme are also useful for source series termination (9 of the 13 ins are outputs from the Pod). See the other sections on Termination and the SIF Slave PCB.

The resistors are :

- 0603, 100 ohm.

The fuses are :

- Raychem Polyswitch fuse. nanoSMDM012.

The protection diodes are :

- Fairchild MMBD1203.

The Transorb is :

- ST SMLVT3V3.

# 10 Electrical

## 10.1 Power Supply

The power supply voltage (SIF_VDD on Pin 1) used for the 16-IDC interface signals should be the same in the SIF Pod and in the SIF Slave PCB. This voltage can be :

- Set on Slave PCB and copied in the Pod
- Set by the Pod and copied or used in the Slave PCB

In the latter case, the Pod voltage is controlled from the Computer. The most common output voltages used in SIF Pods (and available in Xilinx Virtex2 FPGAs) are :

- 3V3
- 2V5

These allow TTL input logic thresholds to be used.

The Virtex2 FPGAs can also support 1V8 and 1V5, but these have not been used in SIF Pods to date.

Old Pods (ISA, LPT) support 5V. This is why the SIF Slave pads need to be 5V-tolerant.

## 10.2 Input Voltage Thresholds

SIF Pads (in Pod and Slave) use TTL voltage thresholds :

| *Logic Value* | *Min* | *Max* |
|---|---|---|
| 0 | 0 V | 0.8 V |
| 1 | 2.0 V | SIF_VDD |

## 10.3 Output Voltage Levels

Output pads in SIF Slave and SIF Pods should conform to the following voltage levels :

| *Logic Value* | *Min* | *Max* |
|---|---|---|
| 0 | 0 V | 0.1 SIF_VDD |
| 1 | 0.9 SIF_VDD | SIF_VDD |

## 10.4 Timing

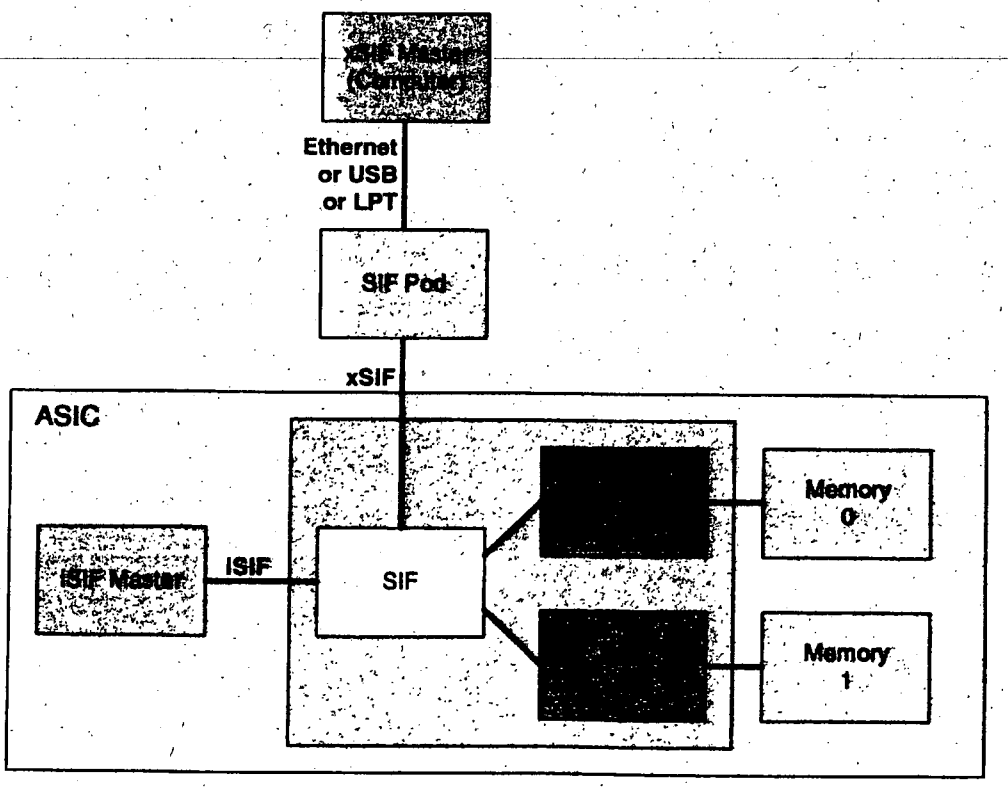
In Command mode, the maximum frequency for SIF_CLK is 1 MHz.

The duty cycle for SIF_CLK must be always between 60:40 and 40:60.

## 11 iSIF – Internal SIF

Previous SIF systems have only provided an external serial physical interface to the SIF Slave. This interface was called SIF but will now be called xSIF (external SIF). A new development to SIF technology is to provide a second physical interface to the SIF Slave called iSIF. iSIF is an internal parallel interface :

| *Name* | *Meaning* | *Type* | *Comment* |
|---|---|---|---|
| xSIF | External SIF | Serial | Same as original SIF, using pins :<br>SIF_LOADB, SIF_CLK, SIF_MOSI, SIF_MISO, SIF_CS (optional).<br>xSIF Master = external computer accessing Slave via a SIF Pod. |
| iSIF | Internal SIF | Parallel | Provides the same functionality as original SIF, but to an on-chip device with parallel bus interfaces to the SIF fields :<br>Address, Control, DataWrite, DataRead, Status.<br>iSIF Master = on-chip hardware outside SIF/XAP. This could be another on-chip processor or a dedicated hardware state machine. |



Operation of the combined xSIF/iSIF obeys the following principles :

- iSIF is optional. It does not have to be connected up in the ASIC. When not used, the iSIF inputs should be tied low and the iSIF outputs left unconnected. When synthesised, almost all the iSIF circuitry will be optimised away.
- Even when used, the addition of iSIF adds very few gates to those originally required for xSIF.
- The SIF instructions available are identical, whether accessed by xSIF or iSIF. This includes Normal and Debug SIF instructions.
- In general, the xSIF and iSIF interfaces are peer to peer. The only area where one interface takes precedence over the other is in the area of SIF Cancel. xSIF may cancel a stuck xSIF or stuck iSIF instruction. iSIF can only cancel a stuck iSIF instruction.
- The designer should use either xSIF or iSIF for sytem debug, but not both.
- The non-debug interface can be used for Normal SIF instructions (i.e to read or write any memory-mapped device).
- The most common usage is that xSIF is used for debug and normal SIF instructions. iSIF is only used for normal SIF instructions, primarily for software download to a XAP RAM.
- iSIF provides a very powerful interface to the iSIF Master (on-chip processor or control hardware outside the XAP/SIF). The iSIF Master can control the XAP as a slave. This functionality must be used with great care by the ASIC designers using XAP/SIF IP (hardware and software engineers). If used wrongly, it could cause complex system errors. When used correctly it provides great functionality for low-power design and precise boot-up control.
- At any one time, the SIF can only process one SIF instruction. This might be from xSIF or from iSIF, but cannot be for both simultaneously.
- SIF instructions are executed at the same time for xSIF and iSIF. SIF Instructions are executed immediately when the selected XAP is stopped or asleep. Otherwise it waits until the next sif instruction executed by the selected XAP processor.
- The XAP can be stopped by the 'Stop' Debug SIF Instruction (from xSIF or iSIF). It can also be stopped by the force_stop input (controlled by on-chip hardware). This could be controlled by the same processor that controls iSIF.
- The SIF has ping-pong hardware to ensure that the SIF cannot be hogged by xSIF or by iSIF. It forces time-multiplexing to share the SIF between xSIF and iSIF. A pending xSIF instruction will be executed when an iSIF instruction is completed. A pending iSIF instruction will be executed when an xSIF instruction is completed.
- iSIF is normally faster than xSIF as it is parallel and on-chip.

## 11.1 Usage Models

SIF instructions are either Normal or Debug instructions :

- Normal instructions can read or write memory.
- Debug instructions can directly control the processor in many ways :
  - Put the processor into Debug or Normal mode
  - read and write registers
  - start and stop the processor
  - single-step the processor
  - run the processor until it hits a break point
  - etc
- Most Debug instructions require the processor to already be in Debug Mode.

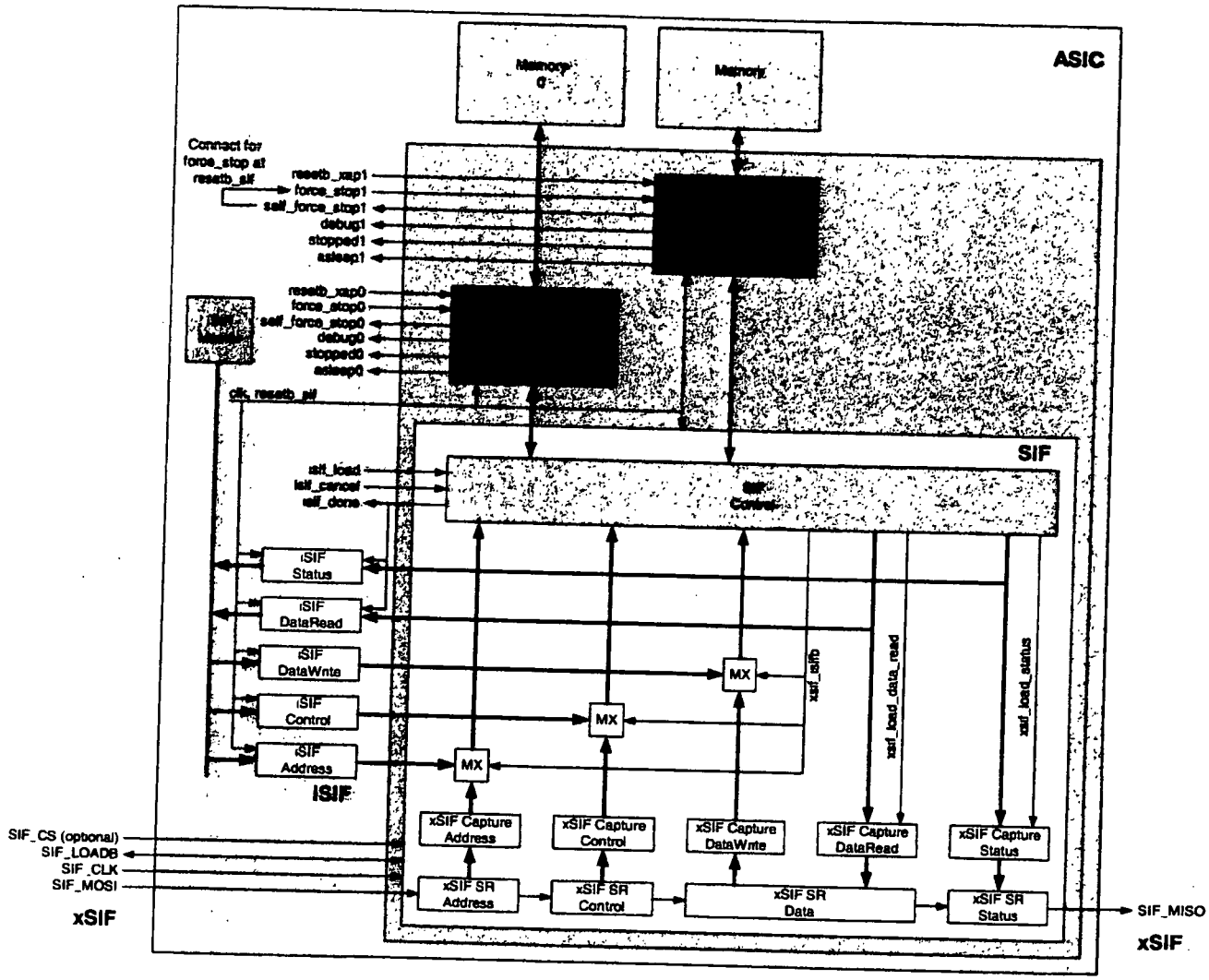
SIF instructions are executed when :

- Processor executes a sif instruction
- Processor is asleep (has executed a sleepsif instruction)
- Processor is stopped (only possible in Debug mode or when force_stop signal is pulled high)

Future ASICs using SIF technology should use xSIF and iSIF in one of the following usage models :

| *No* | *xSIF* | | *iSIF* | | *Comment* |
|---|---|---|---|---|---|
| | Debug | Normal | Debug | Normal | |
| 1 | Yes | Yes | No | No | iSIF not connected or used. |
| 2 | Yes | Yes | No | Yes | iSIF used for memory access only. e.g for program download or data acquisition. |
| 3 | No | No | Yes | Yes | xSIF not connected or used. |
| 4 | No | Yes | Yes | Yes | xSIF used for memory access only. e.g for program download or data acquisition. |

## 11.2 iSIF Hardware Implementation




XAP systems should offer the iSIF and xSIF hardware interfaces shown above. The new signals for iSIF are as follows. All control signals are active-high. All iSIF circuitry and interface signals are clocked on posedge clk and are reset (normally to 0) by the asynchronous reset signal resetb_sif. All outputs are reset to 0.

| *Signal* | *Direction at SIF module* | *Comment* |
|---|---|---|
| isif_address[a:0] | input | Same word format as xSIF Address field. Written to by iSIF Master. |
| isif_control[b:0] | input | Same word format as xSIF Control field. Written to by iSIF Master. Bits indicate – specific processor, debug mode, word size, parity, read/write. Designer may tie some of these bits permanently high or low (e.g to disable Debug SIF Instructions). |
| isif_data_write[c:0] | input | Same word format as xSIF DataWrite field. Written to by iSIF Master. Data to be written to specified address. Data is aligned to bit 0. Unused high bits (for short words) are ignored. |
| isif_data_read[c:0] | output | Same word format as xSIF DataRead field. Read by iSIF Master. Data read from specified address. Data is aligned to bit 0. Unused high bits (for short words) are set to 0. |
| isif_status[d:0] | output | Same word format as xSIF Status field. Read by iSIF Master. Bits indicate – PARITY_S, PARITY_D, ERROR, ERROR_TYPE, ERROR_CODE[], STAT[]. iSIF Master may use or ignore these as it wishes. |
| isif_load | input | SIF starts the iSIF instruction loaded in isif_control[], isif_address[] and isif_data_write[] on posedge clk when isif_load = 1. isif_load should be set to 1 for one clk cycle only and should be 0 at all other times. isif_load may be set by iSIF Master software, or could automatically be set by hardware after iSIF Master writes to isif_control[]. |
| isif_cancel | input | If iSIF Master wants to cancel an iSIF instruction, it does so by setting isif_cancel = 1 for one clk cycle. It should be 0 at all other times. |
| isif_done | output | SIF sets isif_done high for one clk cycle. when the iSIF instruction is complete. The isif_data_read[] and isif_status[] buses are valid on posedge clk when isif_done = 1. isif_done can be used as an enable signal to load these 2 buses into their |

| | | |
|---|---|---|
| | | respective external registers. |

In addition, the following signals should be available for each XAP. These signals are all active-high. They are all sampled and change on posedge clk. The 4 outputs are all reset by resetb_sif. They are all reset to 0, except self_force_stop which is reset to 1. :

| *Signal* | *Direction at XAP module* | *Comment* |
|---|---|---|
| force_stop | input | When force_stop = 1, the XAP completes the current instruction and then stops executing code. If the XAP was executing a block instruction, it completes the current word and then stops execution (i.e same behaviour as for interrupts). The behaviour is the same as when a SIF instruction sets the STOP bit to 1. XAP execution restarts on posedge clk when force_stop = 0 (provided that the SIF has not put the XAP into the Stopped RUN_STATE). |
| self_force_stop | output | Set to 1 by resetb_sif active. resetb_sif is normally pulled low at power-on-reset.<br>Set to 0 by a specific Debug SIF instruction (from xSIF or iSIF).<br>self_force_stop is used by the hardware designer to set the XAP RUN_STATE after resetb_sif :<br>• Stopped : Connect self_force_stop to the force_stop input (or OR with other force_stop sources).<br>• Running : Leave self_force_stop unconnected. |
| debug | output | This is the same as the DEBUG bit in the XAP Status register. It is 1 when the SIF is in Debug mode, and 0 when the SIF is in Normal mode. This is set by xSIF or iSIF Debug Enable writes. |
| stopped | output | This is 0 when the XAP is executing code and 1 when it is stopped. It goes to 1 on posedge clk after it completes the last instruction. This may have been caused by a Debug SIF write or by setting force_stop = 1. It goes to 0 on posedge clk when it starts executing XAP code again.<br>stopped = 1 when any of the following occur : |

| | | |
|---|---|---|
| | | • force_stop = 1<br>• RUN_STATE = Stopped<br>• RUN_STATE = Single Step |
| asleep | output | This is 0 when the XAP is running code and 1 when it is in sleep mode (caused by a sleepnop or sleepsif XAP instruction). It goes to 1 on posedge clk after it completes the sleepnop or sleepsif instruction. It goes to 0 on posedge clk when it starts executing XAP code again (after a hardware wake_up or a SIF Wakeup instruction). |

### 11.3 xSIF v iSIF Priority

If the SIF is currently unused and xSIF and iSIF load requests happen on the same clk cycle, then the xSIF should be served first. Similarly in all equal cases of contention where xSIF and iSIF request the same service simultaneously, then xSIF should be served first.

The SIF Control circuitry implements a ping-pong system to prevent iSIF from blocking xSIF and to prevent xSIF from blocking iSIF :

- If there is an iSIF load request waiting when the SIF completes an xSIF access, then its next access will be for iSIF.
- If there is an xSIF load request waiting when the SIF completes an iSIF access, then its next access will be for xSIF.

This means that the SIF channel bandwidth (whether due to the XAP being stopped or executing sif or sleepsif instructions) is shared by time-multiplexing between xSIF and iSIF.

### 11.4 Parity Bits

The bit format of the main SIF fields is identical for xSIF and iSIF. The Control and Status fields contain parity bits. Their usage by xSIF is described in section 5.3.1 of this TM. In iSIF the parity bits are treated as follows :

| *SIF Field* | *Parity Bit* | *Comment* |
|---|---|---|
| Control | PARITY_AC | iSIF will ignore this. |
| Status | PARITY_D | iSIF will set this. iSIF Master may use this if it wishes. This is unlikely to be necessary, as the signals are all on-chip. |
| Status | PARITY_S | iSIF will set this. iSIF Master may use this if it wishes. This is unlikely to be necessary, as the signals are all on-chip. |

The SIF fields contain parity bits to expose communication errors. This is a valuable feature for xSIF as the off-chip serial interface is exposed to electrical interference.

Parity bits are not that useful for iSIF as the on-chip parallel interface is not exposed to any more electrical interference than the rest of the chip. This is why iSIF ignores PARITY_AC and the iSIF Master can safely ignore PARITY_D and PARITY_S.

### 11.5 SIF Command Mode

SIF Command Mode is available to xSIF but not to iSIF.

## 11.6 SIF Cancel

isif_cancel cancels an iSIF instruction. If there is no iSIF instruction being executed, then isif_cancel is ignored.

When the SIF_CS pin is pulled low, any running xSIF instruction is cancelled. It does not affect any running iSIF instruction.

When SIF_LOADB = 0 and SIF_CLK is toggled 32 times it can cause a SIF Cancel. The last 8 bits on SIF_MOSI are sampled on negedge SIF_CLK to specify the type of SIF Cancel. The decode is only performed after the 32nd negedge SIF_CLK. These bits [7:0] are interpreted as follows :

| *Value* | *Meaning* |
|---|---|
| 0 | Cancel xSIF instruction. |
| 1 | Cancel iSIF instruction. |
| 2 | Cancel xSIF and iSIF instructions. |
| 3 – 255 | Reserved for future use. No action at present. |

## 11.7 XAP Status Register

Each XAP has a read-only Status register that includes several bits.

The iSIF requires the following new bits to be added to each XAP Status Register. These will be present in all XAP processors, even if the iSIF is not being used.

| *Name* | *Meaning* | *Comment* |
|---|---|---|
| FORCE_STOP | Current value on force_stop signal. | Allows xSIF and iSIF to see if the selected XAP is being stopped by force_stop input. |
| FORCE_STOPPED | Set to 1 when force_stop = 1.<br>Set to 0 on clk cycle after iSIF or xSIF write to this XAP Status Reg. | Allows xSIF and iSIF to see if the selected XAP has been stopped by force_stop since the last clear by Status Write. |
| ISIF_ACTIVITY | Set to 1 for the selected XAP when isif_load = 1.<br>Set to 0 on clk cycle after xSIF write to this XAP Status Reg. | Allows xSIF to see if there have been any iSIF accesses to the selected XAP since the last clear by Status Write. |
| XSIF_ACTIVITY | Set to 1 for the selected processor when xsif_load = 1.<br>Set to 0 on clk cycle after iSIF write to this XAP Status Reg. | Allows iSIF to see if there have been any xSIF accesses to the selected XAP since the last clear by Status Write. |

## 11.8 Connecting and using iSIF

If the application designer does not want to use iSIF, he should tie all iSIF inputs to 0. At synthesis, most iSIF hardware will be optimised away.

If the application designer does want to use iSIF, he should choose his strategies :

- Select a device to be iSIF Master. This might be another microprocessor, or could be a hardware state machine.
- Choose whether iSIF is to be used for just Normal SIF instructions, or for Debug SIF instructions as well. Most users only use iSIF for Normal SIF instructions. This is normally used for program download to RAM-based XAP systems.
- If iSIF is used for debug, then xSIF should not be. This is unusual as it requires a new XAP software toolkit (equivalent to xIDE) to be developed.
- Choose whether iSIF access is to be done when :
  - XAP is stopped (by pulling force_stop high).
  - XAP is executing a sleepsif or sif instruction. This gives lower iSIF bandwidth, but does mean that iSIF accesses do not affect the timing or behaviour of the XAP program being run (unless iSIF writes affect its behaviour).
- Choose how much of the iSIF Control field to connect up. It may be safer to tie certain bits to a fixed value (PROCESSOR[], DEBUG, PARITY_AC, WRITE). This can be used to disable certain iSIF features.

The designer should then connect the hardware :

- Connect all iSIF interface signals to registers clocked on posedge clk and reset asynchronously by resetb_sif. If certain signals are not needed (e.g DEBUG bit in isif_control field), they may be tied high or low instead. If the registered signals come from another clock domain, it is recommended that they pass through 2 flipflops clocked by clk, to reduce meta-stability problems. This is especially important for the control signals isif_load and isif_cancel.
- If the XAP cannot execute valid code after reset (e.g because it is a RAM-based system), then force_stop should be pulled high at reset. This will stop the XAP from executing any code until force_stop is pulled low (enabling clean XAP execution) :
  - iSIF control : Connect a flipflop (that resets high) to force_stop XAP input. iSIF Master clears this flipflop to 0 after it has completed code download.
  - xSIF control : Connect self_force_stop XAP output to force_stop input to same XAP. xSIF Master does a Debug SIF write to clear self_force_stop to 0 after it has completed code download.

The control scheme (whether done as software or as hardware state machine) from iSIF Master to SIF should be as follows :

- Pull force_stop high if desired. This will stop the XAP (which may or may not be desirable) and allow faster iSIF accesses. If force_stop is left low and the SIF is in Normal mode, then most SIF instructions require the XAP code to contain some sif or sleepsif instructions.
- For each iSIF access, the iSIF Master should :
  - Write the isif_address field. In Normal mode, this is the address to be read or written.
  - If the iSIF access is a write, the iSIF master should also write the isif_data_write field. This is the data to be written to the selected address.
  - Write the isif_control field. The control field will indicate whether the access is a read or write, the XAP processor to access and the data-size (8, 16 or 32-bits).
  - Set isif_load = 1 for one clk cycle. This could be done automatically by hardware after the isif_control field was updated.
  - Wait for isif_done to go high (which it will for one clk cycle when the iSIF access is completed).
  - If isif_done has still not gone high after a timeout period (selected by the application designer), the iSIF Master should set isif_cancel = 1 for one clk cycle.
  - If isif_done does go high, the iSIF Master should then read the new registered field values :
    - Read the registered isif_data_read field (if the iSIF access is a read).
    - Read the registered isif_status field.

The iSIF Master does not need to write to all the write fields (isif_address, isif_data_write, isif_control) if it knows they already contain the correct values.

Cambridge Consultants

# xSIF API

Prepared by _____

Reviewed by _____

Secretary _____

**Cambridge Consultants Ltd**
Science Park
Milton Road
Cambridge
England CB4 0DW

Tel +44 (0)1223 420024
Fax +44 (0)1223 423373
info@CambridgeConsultants.com
www.CambridgeConsultants.com

C4862-TM-016 v0.10.nc
12 May 2005

**Subject:** xSIF API

## Distribution

Cambridge Consultants

Project Master file

## Cambridge Consultants Ltd – Project Team

Project Reviewer

Project Manager

Technical Contributors

Project Secretary

# Table of Contents


**1 Introduction** 5
1.1 Related Documents 5

**2 System Overview** 6
2.1 16-IDC SIF Interface 6
2.2 SIF Shift Register 7
2.3 SIF Pods and Operating System Support 7
2.4 Operating System Support 7
2.5 Configuration Tool 8

**3 Functional Description** 9
3.1 Atomic Functions 9
3.2 Configuration 9
3.3 Slave ID and Pod ID 10
3.4 Slave and Pod Configuration 11
3.5 Application Configuration 11
3.6 Error Reporting 11
3.7 Reads 11
3.8 Writes 12
3.9 Cycle 12
3.10 Continuous Read 13
3.11 Current Slave 15
3.12 Claim and Release 15
3.13 IO 16

**4 xSIF Application Programmer's Interface** 17
4.1 xSIF Data Types 17
enum `XSIF_ERROR` 17
enum `XSIF_POD_TYPE` 18
enum `XSIF_SLAVE_CS` 18
enum `XSIF_SIGNAL` 19
enum `XSIF_DIAG` 20
structure `xsif_pod` 20
structure `xsif_pod_config` 21
structure `xsif_slave` 22
structure `xsif_slave_config` 23
4.2 xSIF Utility Functions 24
function `xsif_get_version()` 24
function `xsif_get_error_string()` 24
4.3 xSIF Pod Functions 24
function `xsif_init_pod()` 24
function `xsif_get_pods()` 25
function `xsif_get_pod_config()` 26
function `xsif_set_pod_config()` 26
4.4 xSIF Slave Functions 27
function `xsif_get_slaves()` 27
function `xsif_get_slave_config()` 27
function `xsif_set_slave_config()` 28
function `xsif_get_current_slave()` 28

function xsif_set_current_slave() 29
function xsif_get_slave_diag() 29
function xsif_set_slave_diag() 30
4.5 xSIF Shift Register Functions 30
function xsif_read_generic() 30
function xsif_read() 31
function xsif_read_block() 32
function xsif_read_array() 33
function xsif_write_generic() 33
function xsif_write() 34
function xsif_write_block() 35
function xsif_cycle_generic() 36
function xsif_cycle() 36
function xsif_contread_start() 37
function xsif_contread_get() 38
function xsif_contread_done() 39
function xsif_contread_stop() 39
4 6 xSIF IO Functions 40
function xsif_claim() 40
function xsif_release() 40
function xsif_get_io_dir() 41
function xsif_set_io_dir() 41
function xsif_get_io() 42
function xsif_set_io() 42

# 1 Introduction

This document describes the xSIF Drivers and the xSIF API. The API is available to C/C++ and Matlab.

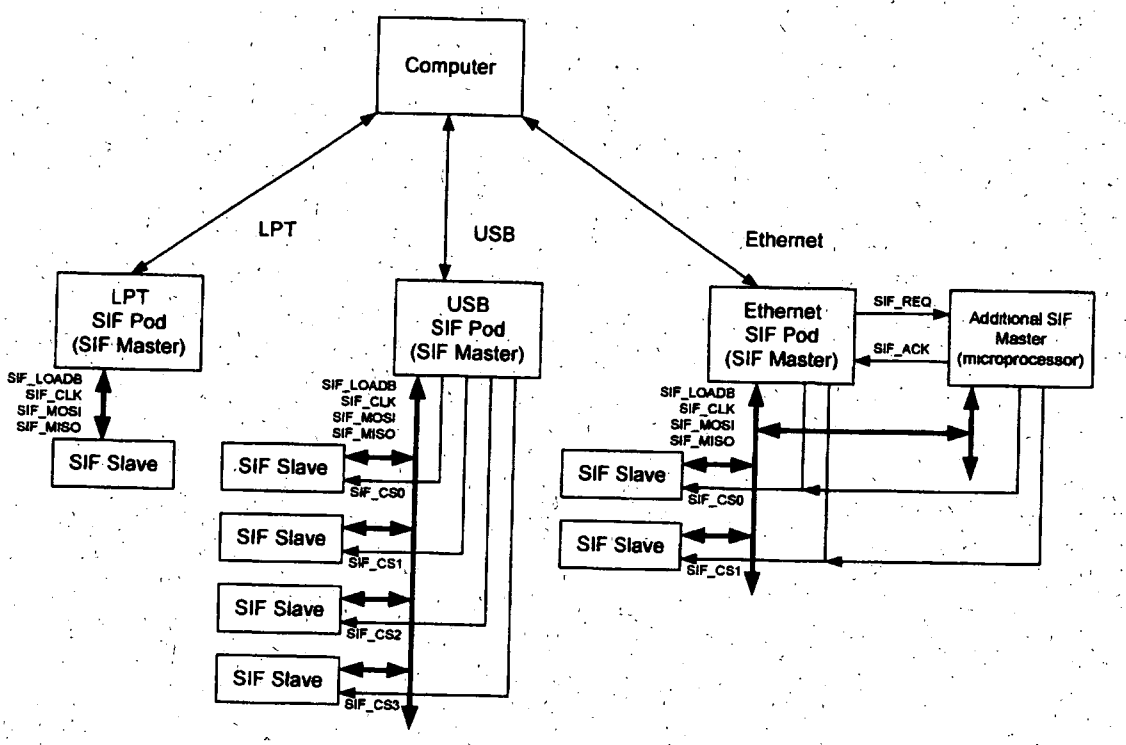
The xSIF Drivers provide a common interface from SIF Masters to SIF Pods and SIF Slaves.

The programmer's interface is common to all operating systems.

## 1.1 Related Documents

# 2 System Overview

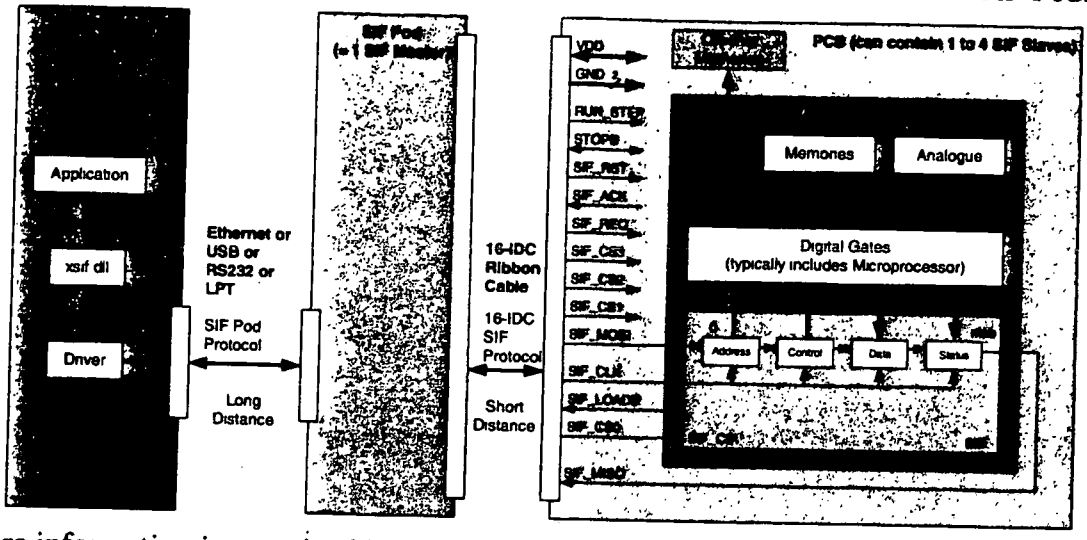
The diagram below shows the components of a SIF system. More information is contained in C4862-TM-005.




## 2.1 16-IDC SIF Interface

The drivers provide an interface to a standard 16-IDC connector. This connector is common to all SIF Pods.

The diagram below shows how a computer is connected to a SIF slave via a SIF Pod.




More information is contained in C4862-TM-005.

## 2.2 SIF Shift Register

The SIF shift register has 4 fields.

- Address
- Control
- Data
- Status

The lengths of these fields are fixed for a particular slave. The lengths are specified separately and can be any value from 0 to 32 bits. The API functions align words with their least significant bit at bit zero.

## 2.3 SIF Pods and Operating System Support

The xSIF Drivers support three types of SIF Pod.

- LPT SIF Pod
- USB SIF Pod
- Universal SIF Pod (Ethernet, USB and Serial)

## 2.4 Operating System Support

The xSIF Drivers are available for the following operating systems.

- Windows 2000

- Windows XP
- x86 Linux
- Mac OS X

The table below shows which combinations of operating systems and pods are supported.

| | Windows 2000/XP | x86 Linux | Mac OS X |
|---|---|---|---|
| Universal Pod (Ethernet) | Yes | Yes | Yes |
| Universal Pod (Serial) | Yes | Yes | Yes |
| Universal Pod (USB) | Yes | No | No |
| USB Pod | Yes | No | No |
| LPT Pod | Yes | No | No |

## 2.5 Configuration Tool

The xSIF drivers include a graphical configuration tool for configuring pods and slaves.

# 3 Functional Description

## 3.1 Atomic Functions

All xSIF functions are atomic.

From the pod's point of view, when a function is completed it has no more SIF activity to perform. The exception to this is when the continuous buffer is running, in which case the pod will continue to do SIF reads.

## 3.2 Configuration

Pods and slaves can be used by multiple applications. Applications can use more than one slave at a time.

The slave and pod configuration is shared by all applications using xSIF. The configuration information is split into fixed and variable parts.

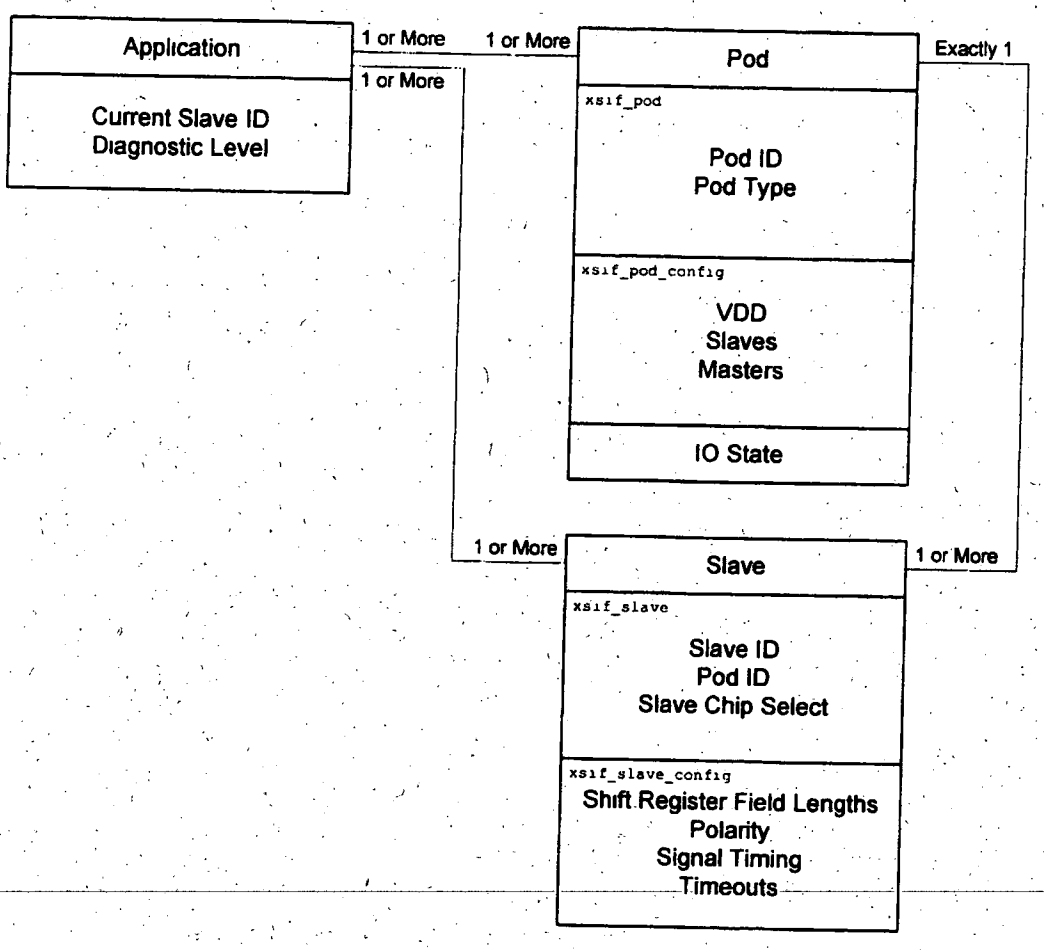
The API supports changing the variable part of the configuration information. Changes to this information affect all other applications using xSIF. This information is stored by the operating system as persistent application settings.

The fixed configuration information cannot be changed by using functions in the API.

The table below shows the data types used to hold the configuration information in the system.

| | Fixed | Variable |
|---|---|---|
| Slave | `xsif_slave` | `xsif_slave_config` |
| Pod | `xsif_pod` | `xsif_pod_config` |

The diagram below shows the relationship between applications, pods and slaves.




## 3.3 Slave ID and Pod ID

Each Pod is assigned a globally unique `pod_id` and each slave is assigned a globally unique `slave_id`.

The drivers have the concept of Current Slave. Many of the functions in the API act upon the current slave rather than specifying the slave in each call.

This API does not support the assignment of `pod_id` and `slave_id` values. This is performed using the xSIF Configuration Tool. When the drivers detect new SIF Pods, new `pod_id` and `slave_id` values will be created. The existing `pod_id` and `slave_id` values will not be changed.

When an application starts, the current slave is set to the value of the initial slave specified in the xSIF Configuration Tool. This is the initial slave used by the application until another slave is selected.

## 3.4 Slave and Pod Configuration

The API allows users to change the settings of a Pod and Slave. The Pod settings affect all of the slaves connected to the Pod. The slave settings affect the individual slave.

Making changes to the slave and pod configurations affects all other applications using the slaves and pods. Applications will not normally need to change the slave and pod configuration.

## 3.5 Application Configuration

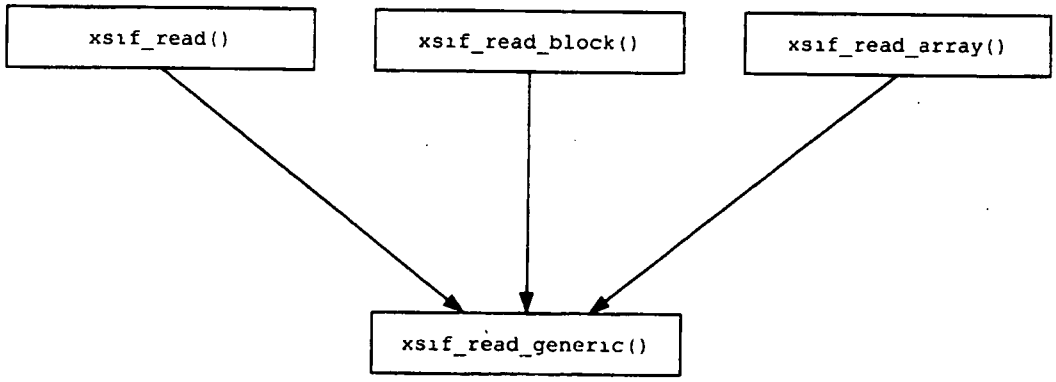
Each application using the drivers has state information that is independent of other applications. This includes current slave and level of diagnostics.

## 3.6 Error Reporting

Most of the functions in the API return a value indicating success or failure. The function `xsif_get_error_string()` can be called to get more detailed information about the error.

## 3.7 Reads

The diagram below shows the read functions in the API. The `xsif_read_generic()` function is used to implement all the higher level functions. Users will normally not call `xsif_read_generic()`, but one of the convenience functions shown.




The `xsif_read()` function reads a single value.

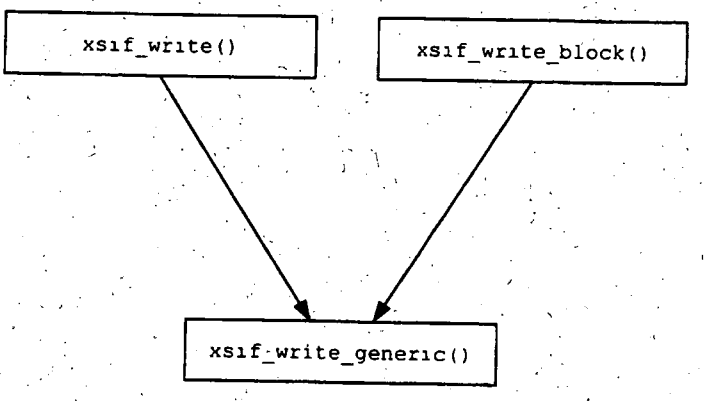
The `xsif_read_block()` function reads multiple values from a sequence of addresses. Users specify the starting address, the amount to increase the address by and how many addresses to read. This is useful for verifying program memory when loading programs.

The `xsif_read_array()` function reads a sequence of addresses multiple times. Users specify a list of addresses to read and the total number of reads to perform. This

function is useful for gathering data to characterise hardware such as multiple ADC channels.

## 3.8 Writes

The diagram below shows the write functions in the API. The `xsif_write_generic()` function is used to implement all the higher level functions. Users will normally not call `xsif_write_generic()`, but one of the convenience functions shown.




The `xsif_write()` function writes a single value.

The `xsif_write_block()` function writes multiple values to a sequence of addresses. Users specify the starting address, the amount to increase the address by and how many addresses to write. This is useful for loading programs into memory.

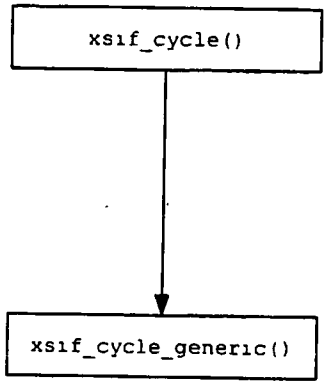
## 3.9 Cycle

A cycle is a generic SIF access routine that can be used to implement all SIF accesses. SIF read and SIF writes can be implemented by performing a sequence of SIF cycles.

A SIF cycle consists of two phases. The first phase is to shift a word from the master to the slave while receiving the bits shifted from the slave to the master. The second phase is an optional `SIF_LOADB` handshake.

The word lengths and the write bit are under the control of the programmer.

The diagram below shows the two cycle functions.




### 3.10 Continuous Read

The functions supporting continuous reads are only available with USB and Ethernet Pods.

The SIF Drivers are able to perform continuous data acquisition from single or multiple SIF addresses. When the continuous read is running, the pod automatically performs SIF accesses and the driver buffers the data. The application is responsible for processing blocks of data from the buffer to avoid overflow.

The addresses to read are specified in exactly the same way as with the `xsif_read_array()` function. The continuous read can be thought of as an `xsif_read_array()` operation where the number of reads is unknown and the data is returned in blocks.

The application specifies the size of the blocks of data and status words it would like. When the continuous read is in progress the application polls to see if new data and status words are available. The application gets pointers to the new data and status words when they are ready. The application must then process the new data and indicate to the driver when it has finished.

The driver is not allowed to write to the block of data that the application is processing and the application is not allowed to write to a block of data that it has told the driver it has finished processing.

There are four functions associated with the continuous buffer.

The `xsif_contread_start()` function is used to specify a list of addresses and begins the data acquisition.

Applications use the `xsif_contread_get()` function to ask the driver if a new block of data and status words is ready. The application is returned pointers to new data and status words if they are available. Calls to `xsif_contread_get()` also return a word to

indicate whether the buffer has overrun. An overrun is when the application has not processed data fast enough to keep up with the SIF Pod.

The `xsif_contread_done()` function indicates to the driver that the application has finished processing the data and status words from the preceding `xsif_contread_get()` function. The application is no longer allowed to access the data and status words following a call `to xsif_contread_done()`.

The `xsif_contread_stop()` function is used to stop the buffered reads.

The Pod takes advantage of the SIF shift register so that SIF read accesses can be overlapped. The next address and control is shifted in at the same time as the previous data and status is shifted out.

### Example

The following pseudo code illustrates how users will call these functions.

```
// Start acquisition.
xsif_contread_start(address_list, num_reads)

// Poll for new data
while (still_running)
    if (xsif_contread_get())
        Read and process new data
        // Signal that we have finished with the data
        xsif_contread_done()
        Other application tasks

// Stop acquisition
xsif_contread_stop()
```

### Example Usage

A typical PC application will be using the data from the continuous read for two reasons.

- GUI update
- Data acquisition/logging

The requirement of the GUI update is to let the user know the data acquisition is still running and provide an indication of the results. The data acquisition/logging part requires all samples to be processed and some results stored.

## 3.11 Current Slave

The `xsif_set_current_slave()` function is used to select the Current Slave. Changing the current slave does not effect other applications using the same SIF Pod. This allows more than one Application to communicate with the same or different slaves using the same SIF Pod.

When a slave associated with a chip select signal is chosen, the chip select signal to that SIF slave will be asserted. If the SIF is claimed the new chip select signal is asserted immediately.

If the SIF is released when `xsif_set_current_slave()` is called, the new chip select is asserted the next time the SIF is claimed.

The chip select signals are shared with some of the general IO signals; signals that are not being used as chip select signals are available as general IO.

## 3.12 Claim and Release

The SIF signals may be 'claimed' and 'released' using the `xsif_claim()` and `xsif_release()` functions, respectively. When claimed the SIF signals and any chip selects are enabled and ready to perform SIF accesses. When released the SIF signals and any chip selects are tri-stated.

The claim and release procedure is dependent on whether there are multiple slaves and whether there are multiple masters. The SIF drivers must be correctly configured.

When there are multiple masters the arbitration procedure is performed when `sif_claim()` is called. If the bus is not available `xsif_claim()` returns an error code. The arbitration signals are shared with some of the general IO signals; when multiple masters are used `SIF_ACK` and `SIF_REQ` are no longer available as general IO.

If the SIF is released and a SIF access function is called, the drivers bracket the access with calls to `xsif_claim()` and `xsif_release()`. By keeping the SIF claimed, applications can increase the data rate achieved by SIF. When there is more than one application using a SIF Pod and there are Multiple Masters it is suggested that `xsif_claim()` and `xsif_release()` are not used directly. The drivers will safely bracket SIF accesses with `xsif_claim()` and `xsif_release()`.

When there is a Single Master the drivers automatically call the `xsif_claim()` function during initialisation. This means that the SIF signals are permanently driven when there is a Single Master. Users may still make calls to `xsif_claim()` and `xsif_release()` in order to control whether the SIF signals and chip selects are driven or tristated. It is simpler to leave the signals permanently driven (or claimed), however, tristated SIF signals are useful when measuring the power consumption of a slave.

## 3.13 IO

Unused signals on the SIF connector are available as general purpose IO signals using the xSIF IO functions.

The general purpose IO signals can independently be configured as input or output signals in the USB and Ethernet Pods

The direction of the general purpose IO signals cannot be changed in LPT Pods. It is, also, not possible to independently enable and disable all of the output signals in the LPT Pod. C4862-TM-003 contains more information about the IO capabilities of the LPT Pod.

# 4 xSIF Application Programmer's Interface

## 4.1 xSIF Data Types

---

### *enum* XSIF_ERROR

Most xSIF functions return an error code of type XSIF_ERROR.

```
typedef enum
{
    XSIF_OK = 0,
    XSIF_GENERAL,
    XSIF_INVALID,
    XSIF_LOADB_LOW_TIMEOUT,
    XSIF_LOADB_HIGH_TIMEOUT,
    XSIF_LOADB_LOW,
    XSIF_ACK_LOW_TIMEOUT,
    XSIF_ACK_HIGH_TIMEOUT,
    XSIF_UNKNOWN_POD,
    XSIF_UNKNOWN_SLAVE,
} XSIF_ERROR;
```

**Values**

| | |
|---|---|
| XSIF_OK | Function completed successfully |
| XSIF_GENERAL | General purpose error code. |
| XSIF_INVALID | Invalid arguments passed to the function or function not supported. |
| XSIF_LOADB_LOW_TIMEOUT | SIF Shift Register function timed out because SIF_LOADB was stuck low. |
| XSIF_LOADB_HIGH_TIMEOUT | SIF Shift Register function timed out because SIF_LOADB was stuck high. |
| XSIF_LOADB_LOW | SIF Shift Register function was not able to start, because SIF_LOADB was already low. |
| XSIF_ACK_LOW_TIMEOUT | Function timed out when claiming or releasing the SIF because SIF_ACK was stuck low. |
| XSIF_ACK_HIGH_TIMEOUT | Function timed out when claiming or releasing the SIF because SIF_ACK was stuck high. |

| | |
|---|---|
| XSIF_UNKNOWN_POD | Pod ID is not valid. |
| XSIF_UNKNOWN_SLAVE | Slave ID is not valid. |

---

### *enum* XSIF_POD_TYPE

The type of a Pod.

```
typedef enum
{
    XSIF_LPT = 0,
    XSIF_ISA98,
    XSIF_ISA94,
    XSIF_USB,
    XSIF_ETHERNET,
    XSIF_UNIVERSAL,
} XSIF_POD_TYPE;
```

**Values**

| | |
|---|---|
| XSIF_LPT | LPT Pod (printer port) |
| XSIF_ISA98 | ISA Card plus Pod 98 |
| XSIF_ISA94 | ISA Card plus Pod 94 |
| XSIF_USB | USB Pod |
| XSIF_ETHERNET | Ethernet Pod |
| XSIF_UNIVERSAL | Universal Pod |

---

### *enum XSIF_SLAVE_CS*

This type identifies a slave or collection of slaves. To identify a particular slave connected to a pod use one of the enumerated values. To identify a collection of slaves connected to a pod OR together the XSIF_SLAVE_CS values.

```
typedef enum
{
    XSIF_SLAVE_NO_CS = 0x00000000,
    XSIF_SLAVE_CS0 = 0x00000001,
    XSIF_SLAVE_CS1 = 0x00000002,
    XSIF_SLAVE_CS2 = 0x00000004,
    XSIF_SLAVE_CS3 = 0x00000008,
    XSIF_SLAVE_CS4 = 0x00000010,
```

```
    ...
    XSIF_SLAVE_CS31 = 0x80000000,
} XSIF_SLAVE_CS;
```

**Values**

| | |
|---|---|
| XSIF_SLAVE_NO_CS | When used in the `slave_cs` element of the `xsif_pod_config` or `xsif_pod` structure, this value indicates that the pod is connected to a single slave and no chip selects are used.<br>When used in the `slave_cs` element of the `xsif_slave` structure, this value indicates that the slave is the only slave connected to the pod and that it does not have a chip select signal. |
| XSIF_SLAVE_CS0 | The slave enabled by SIF_CS0. |
| XSIF_SLAVE_CS1 | The slave enabled by SIF_CS1. |
| XSIF_SLAVE_CS2 | The slave enabled by SIF_CS2. |
| XSIF_SLAVE_CS3 | The slave enabled by SIF_CS3. |
| XSIF_SLAVE_CS4 | Reserved for future use. |
| ... | |
| XSIF_SLAVE_CS31 | Reserved for future use. |

---

### *enum* XSIF_SIGNAL

The values in this enumeration are used by the IO functions.

```
typedef enum
{
    // These are the SIF signals on the 16-IDC.
    XSIF_SIGNAL_SIF_MISO = 2,
    XSIF_SIGNAL_SIF_MOSI = 4,
    XSIF_SIGNAL_SIF_CLK = 6,
    XSIF_SIGNAL_SIF_LOADB = 9,
    // These are the general IO signals on the 16-IDC.
    XSIF_SIGNAL_SIF_CS0 = 3,
    XSIF_SIGNAL_SIF_CS1 = 8,
    XSIF_SIGNAL_SIF_CS2 = 10,
    XSIF_SIGNAL_SIF_CS3 = 11,
    XSIF_SIGNAL_SIF_REQ = 12,
```

```
    XSIF_SIGNAL_SIF_ACK = 13,
    XSIF_SIGNAL_SIF_RST = 14,
    // These are the XAP1 signals on the 16-IDC
    XSIF_SIGNAL_STOPB = 15,
    XSIF_SIGNAL_RUN_STEP = 16,
    // These are internal signals for controlling RUN_STEP and STOPB
    XSIF_SIGNAL_PC_STOPPED,          // input
    XSIF_SIGNAL_PC_ENA_STOPPING,     // output
    XSIF_SIGNAL_PC_GO,               // output
    XSIF_SIGNAL_PC_FORCE_STOP,       // output
    // An internal signal to globally enable/disable IO
    XSIF_SIGNAL_ENABLE_ALL           // output
} XSIF_SIGNAL;
```

## *enum* **XSIF_DIAG**

The level of diagnostic testing performed by the driver.

```
typedef enum
{
    XSIF_DIAG_NONE = 0,
    XSIF_DIAG_SINGLE_SIF,
    XSIF_DIAG_MULTI_SIF,
} XSIF_DIAG;
```

### Values

| | |
|---|---|
| XSIF_DIAG_NONE | No diagnostics performed. |
| XSIF_DIAG_SINGLE_SIF | Perform diagnostic check on single SIF Reads or Writes. |
| XSIF_DIAG_MULTI_SIF | Perform diagnostics on all SIF Reads and Writes except those done as part of the continuous buffer. |

## *structure* **xsif_pod**

Fixed information about a Pod.

```
typedef struct xsif_pod
{
    xuint32 pod_id;
    xuint32 status;
```

```
    XSIF_POD_TYPE pod_type;
    xuint32 max_sif_clk_freq;
    xuint32 max_num_addr;
    xuint32 max_contread;
    xuint32 max_slave_cs;
    const char* descr;
} xsif_pod
```

### Elements

| | |
|---|---|
| pod_id | A globally unique id for this Pod. This value selected automatically by the drivers, but can be changed using the Configuration Tool. |
| status | 0 = not present<br>1 = present and functioning<br>2 = present, but unavailable |
| pod_type | The type of Pod. |
| max_sif_clk_freq | The maximum SIF_CLK frequency supported by the Pod in Hz. |
| max_num_addr | The maximum number of addresses that can be specified in the SIF Shift Register functions. |
| max_contread | The maximum number of continuous read operations that can be active at one time. |
| max_slave_cs | Which SIF_CS signals are available for configuration as slave chip select signals. This is a bit field. OR together values from XSIF_SLAVE_CS. |
| descr | A string describing the Pod. This string is entered using the Configuration Tool. |

---

## *structure* xsif_pod_config

The configuration of a Pod that is common to all slaves attached to the Pod. This configures how the 16-IDC pins are to be used.

```
typedef struct xsif_pod_config
{
    xuint32 multi_master;
    xuint32 slave_cs;
    xuint32 vdd_direction;
```

```
    xuint32 vdd_mv;
} xsif_pod_config
```

**Elements**

| | |
|---|---|
| multi_master | Whether the pod is connected to a SIF with single master or to multiple masters.<br>0 = single master<br>1 = multiple masters |
| slave_cs | Which SIF_CS signals are to be used as slave chip selects. This is a bit field. OR together values from XSIF_SLAVE_CS. |
| vdd_direction | The direction of the VDD signal.<br>0 = output from Pod, input to target<br>1 = input to Pod, output from target |
| vdd_mv | This controls the voltage on VDD when VDD is an output from the Pod. The voltage is specified in mV. This value is ignored when VDD is an input. |

---

***structure* xsif_slave**

Fixed information about a Slave.

```
typedef struct xsif_slave
{
    xuint32 slave_id;
    xuint32 pod_id;
    XSIF_SLAVE_CS slave_cs;
    const char* descr;
} xsif_pod
```

**Elements**

| | |
|---|---|
| slave_id | A globally unique id for this Slave. This value selected automatically by the drivers, but can be changed using the Configuration Tool. |
| pod_id | The globally unique id for the Pod to which this Slave is connected. |
| slave_cs | Identifies the chip select from the Pod to the Slave. |
| descr | A string describing the Slave. This string is entered using the Configuration Tool. |

## *structure* `xsif_slave_config`

Configuration of a slave's interface from a Pod. This controls the way a pod communicates with a slave.

```
typedef struct xsif_slave_config
{
    xuint32 address_size;
    xuint32 control_size;
    xuint32 data_size;
    xuint32 status_size;
    xuint32 write_sense;
    xuint32 sif_clk_freq;
    xuint32 loadb_time;
    xuint32 loadb_timeout;
    xuint32 ack_timeout;
} xsif_slave_config
```

### Elements

| | |
|---|---|
| `address_size` | Number of bits in the address word. |
| `control_size` | Number of bits in the control word. This includes the write bit. |
| `data_size` | Number of bits in the data word. |
| `status_size` | Number of bits in the status word. |
| `write_sense` | Polarity of the write bit.<br>0 = Active low write<br>1 = Active high write |
| `sif_clk_freq` | The frequency of `SIF_CLK` in Hz. |
| `loadb_time` | The length of time in `SIF_CLK` periods that the `SIF_LOADB` signal is held low by the master during a handshake. |
| `loadb_timeout` | The length of the timeout in `SIF_CLK` periods when waiting for `SIF_LOADB`. |
| `ack_timeout` | The length of the timeout in `SIF_CLK` periods when waiting for `SIF_ACK`. |

## 4.2 xSIF Utility Functions

### *function* `xsif_get_version()`

Returns the version number of the driver.

#### C Function

```
xuint32 xsif_get_version(void)
```

#### Matlab Function

```
[version] = xsif_get_version()
```

#### Parameters

| | |
|---|---|
| `version` | The returned version number. The version number is coded in the least significant three bytes.<br>`version[7:0]` = bug fix release number<br>`version[15:8]` = minor release number<br>`version[23:16]` = major release number<br>`version[31:24]` = 0 |

### *function* `xsif_get_error_string()`

Returns a string description of an error code returned by an xSIF function.

#### C Function

```
const char* xsif_get_error_string(XSIF_ERROR error)
```

#### Matlab Function

```
[string] = xsif_get_error_string(error)
```

#### Parameters

| | |
|---|---|
| `error` | Returned code from an xSIF function. |

## 4.3 xSIF Pod Functions

### *function* `xsif_init_pod()`

Initialises a SIF Pod to a defined state.

All signals that can be tri-stated are configured as high impedance. SIF_LOADB is high and SIF_CLK is low. All other output signals are low.

### C Function

```
XSIF_ERROR xsif_init_pod(xuint32 pod_id)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_init_pod(pod_id)
```

### Parameters

| | |
|---|---|
| pod_id | Identifies the Pod. |

### Error Conditions

XSIF_OK if successful.

---

## *function* xsif_get_pods()

Returns details of all connected pods. The calling application allocates an array of xsif_pod structures. The number of items allocated is sent to the function in the num_pods argument. The function builds a list of no more than num_pods pod descriptors and returns the number of pods in the num_pods argument. The size argument is sizeof(xsif_pod), this supports the addition of new parameters in the future while remaining binary compatible with old applications.

If pod is set to NULL, num_pods will be written to with the number of pods. This allows applications to use xsif_get_pods() first with pod set to NULL to establish how many pods are present, before allocating memory for the second call to xsif_get_pods(), which passes the allocated memory to receive the pod descriptors.

When pod is NULL, the value of num_pods will be ignored.

Note: the Matlab function does not use the num_pods argument. The function simply returns details of all pods.

### C Function

```
XSIF_ERROR xsif_get_pods(xuint32 size, xsif_pod* pod, xuint32*
num_pods)
```

### Matlab Function

```
[pod, XSIF_ERROR] = xsif_get_pods()
```

### Parameters

| | |
|---|---|
| size | sizeof(xsif_pod) |
| pod | An array of xsif_pod returned by the function. |
| num_pods | The number of pods. This parameter is used as both an input and output from the function. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_get_pod_config()`

Gets the configuration of a SIF pod. This function returns information for a pod and is common to all slaves attached to the Pod.

### C Function

```
XSIF_ERROR xsif_get_pod_config(xuint32 pod_id, xuint32 size,
xsif_pod_config* pod_config)
```

### Matlab Function

```
[pod_config, XSIF_ERROR] = xsif_get_pod_config(pod_id)
```

### Parameters

| | |
|---|---|
| `pod_id` | Identifies the Pod. |
| `size` | `sizeof(xsif_pod_config)` |
| `pod_config` | Pointer to an `xsif_pod_config` structure that will be written to with the current configuration. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_set_pod_config()`

Changes the configuration of the current SIF pod. The configuration information is common to all slaves attached to the pod.

### C Function

```
XSIF_ERROR xsif_set_pod_config(xuint32 pod_id, xuint32 size,
xsif_pod_config* pod_config)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_set_pod_config(pod_id, pod_config)
```

### Parameters

| | |
|---|---|
| `pod_id` | Identifies the Pod. |
| `size` | `sizeof(xsif_pod_config)` |
| `pod_config` | Pointer to an `xsif_pod_config` structure defining the new configuration |

### Error Conditions

XSIF_OK if successful.

## 4.4 xSIF Slave Functions

---

### *function* xsif_get_slaves()

Returns details of all connected slaves. The calling application allocates an array of xsif_slave structures. The number of items allocated is sent to the function in the num_slaves argument. The function builds a list of no more than num_slaves slave descriptors and returns the number of slaves in the num_slaves argument. The size argument is sizeof(xsif_slave), this supports the addition of new parameters in the future while remaining binary compatible with old applications.

If slave is set to NULL, num_slaves will be written to with the number of slaves. This allows applications to use xsif_get_slaves() first with slave set to NULL to establish how many slaves are present, before allocating memory for the second call to xsif_get_slaves(), which passes the allocated memory to receive the slave descriptors.

When slave is NULL, the value of num_slaves will be ignored.

Note: the Matlab function does not use the num_slaves argument. The function simply returns details of all slaves.

#### C Function

```
XSIF_ERROR xsif_get_slaves(xuint32 size, xsif_slave* slave, xuint32*
num_slaves)
```

#### Matlab Function

```
[slave, XSIF_ERROR] = xsif_get_slaves()
```

#### Parameters

| | |
|---|---|
| size | sizeof(xsif_slave) |
| slave | An array of xsif_slave returned by the function. |
| num_slaves | The number of slaves. This parameter is used as both an input and output from the function. |

#### Error Conditions

XSIF_OK if successful.

---

### *function* xsif_get_slave_config()

Gets the configuration of a slave.

### C Function

```
XSIF_ERROR xsif_get_slave_config(xuint32 slave_id, xuint32 size,
xsif_slave_config* slave_config)
```

### Matlab Function

```
[slave_config, XSIF_ERROR] = xsif_get_slave_config(slave_id)
```

### Parameters

| | |
|---|---|
| `slave_id` | Identifies the slave. |
| `size` | `sizeof(xsif_slave_config)` |
| `slave_config` | Pointer to an `xsif_slave_config` structure that will be written to with the current configuration. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_set_slave_config()`

Changes the configuration of a slave.

### C Function

```
XSIF_ERROR xsif_set_slave_config(xuint32 slave_id, xuint32 size,
xsif_slave_config* slave_config)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_set_slave_config(slave_id, slave_config)
```

### Parameters

| | |
|---|---|
| `slave_id` | Identifies the slave. |
| `size` | `sizeof(xsif_slave_config)` |
| `slave_config` | Pointer to an `xsif_slave_config` structure defining the new configuration |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_get_current_slave()`

Gets the ID of the current slave.

### C Function

```
XSIF_ERROR xsif_get_current_slave(xuint32* slave_id)
```

### Matlab Function

```
[slave_id, XSIF_ERROR] = xsif_get_current_slave()
```

### Parameters

| | |
|---|---|
| `slave_id` | Globally unique identifier for a slave. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_set_current_slave()`

Changes the application's Current Slave.

If the new Slave is connected to a Pod that is in the claimed state and the new Slave is connected to a chip select signal then the chip select signal is asserted immediately.

If the new slave is connected to a Pod that is in the released state then the chip select is asserted when the SIF is next claimed.

This function does not claim or release any of the SIF Pods.

### C Function

```
XSIF_ERROR xsif_set_current_slave(xuint32 slave_id)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_set_current_slave(slave_id)
```

### Parameters

| | |
|---|---|
| `slave_id` | Globally unique identifier for a slave. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_get_slave_diag()`

Gets a value showing the level of diagnostics being performed when communicating with the current slave.

### C Function

```
XSIF_ERROR xsif_get_slave_diag(XSIF_DIAG* level)
```

### Matlab Function

```
[level, XSIF_ERROR] = xsif_get_slave_diag()
```

### Parameters

| | |
|---|---|
| `level` | A value from the `XSIF_DIAG` enumeration indicating the diagnostic level. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_set_slave_diag()`

Sets the level of diagnostics performed when communicating with the current slave. When diagnostics are enabled, the Pod will do additional shifts to check the integrity of the shift register.

### C Function

```
XSIF_ERROR xsif_set_slave_diag(XSIF_DIAG level)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_set_slave_diag(level)
```

### Parameters

| | |
|---|---|
| `level` | A value from the `XSIF_DIAG` enumeration indicating the diagnostic level. |

### Error Conditions

`XSIF_OK` if successful.

## 4.5 xSIF Shift Register Functions

---

## *function* `xsif_read_generic()`

This function performs a set of SIF read operations using the current slave. This function is equivalent to the following pseudo code.

```
num_loops = num_reads/num_address
for loop = 1 to num_loops
    offset = address_step * (loop-1)
    for each address
        SIF Read from (address + offset)
```

### C Function

```
XSIF_ERROR xsif_read_generic(xuint32 num_address, xuint32* address,
xuint32 address_step, xuint32 control, xuint32 num_reads, xuint32*
data, xuint32* status)
```

### Matlab Function

```
[data, status, XSIF_ERROR] = xsif_read_generic(address, address_step,
control, num_reads)
```

### Parameters

| | |
|---|---|
| `num_address` | Number of addresses given in the `address` parameter. |
| `address` | An array of addresses that will be read from in sequence. |
| `address_step` | A modifier added to each address in the sequence. |
| `control` | The control word, excluding the write bit. The same control word is used for all reads. The write bit is inserted automatically. |
| `num_reads` | The total number of reads to perform. This must be a multiple of `num_address`. |
| `data` | An array of returned data words. No data is returned if `data` is zero. |
| `status` | An array of returned status words. No status words are returned if `status` is zero. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_read()`

Performs a single SIF read.

This function is provided for convenience and is equivalent to `xsif_read_generic(1, &address, 0, control, 1, data, status)`.

### C Function

```
XSIF_ERROR xsif_read(xuint32 address, xuint32 control, xuint32* data,
xuint32* status)
```

### Matlab Function

```
[data, status, XSIF_ERROR] = xsif_read(address, control)
```

### Parameters

| | |
|---|---|
| address | The address to read. |
| control | The control word, excluding the write bit. The write bit is inserted automatically. |
| data | The returned data word. No data is returned if data is zero. |
| status | The returned status word. No status word is returned if status is zero. |

### Error Conditions

XSIF_OK if successful.

---

## *function* xsif_read_block()

Performs a set of SIF reads from series of addresses separated by a fixed step.

This function is provided for convenience and is equivalent to xsif_read_generic(1, &address, address_step, control, num_reads, data, status).

### C Function

```
XSIF_ERROR xsif_read_block(xuint32 address, xuint32 address_step,
xuint32 control, xuint32 num_reads, xuint32* data, xuint32* status)
```

### Matlab Function

```
[data, status, XSIF_ERROR] = xsif_read_block(address, address_step,
control, num_reads)
```

### Parameters

| | |
|---|---|
| address | The starting address. |
| address_step | A modifier added to the address after each read. |
| control | The control word, excluding the write bit. The same control word is used for all reads. The write bit is inserted automatically. |
| num_reads | The total number of reads to perform. |
| data | An array of returned data words. No data is returned if data is zero. |
| status | An array of returned status words. No status words are returned if status is zero. |

### Error Conditions

XSIF_OK if successful.

## *function* `xsif_read_array()`

Performs a set of SIF reads from a series of addresses a multiple number of times.

This function is provided for convenience and is equivalent to `xsif_read_generic(num_address, address, 0, control, num_reads, data, status)`.

### C Function

```
XSIF_ERROR xsif_read_array(xuint32 num_address, xuint32* address,
xuint32 control, xuint32 num_reads, xuint32* data, xuint32* status)
```

### Matlab Function

```
[data, status, XSIF_ERROR] = xsif_read_array(address, control,
num_reads)
```

### Parameters

| | |
|---|---|
| `num_address` | Number of addresses given in the `address` parameter. |
| `address` | An array of addresses that will be read from in sequence. |
| `control` | The control word, excluding the write bit. The same control word is used for all reads. The write bit is inserted automatically. |
| `num_reads` | The total number of reads to perform. This must be a multiple of `num_address`. |
| `data` | An array of returned data words. No data is returned if `data` is zero. |
| `status` | An array of returned status words. No status words are returned if `status` is zero. |

### Error Conditions

`XSIF_OK` if successful.

## *function* `xsif_write_generic()`

This function performs a set of SIF write operations using the current slave. This function is equivalent to the following pseudo code.

```
num_loops = num_writes/num_address
for loop = 1 to num_loops
    offset = address_step * (loop-1)
    for each address
        SIF Write to (address + offset)
```

### C Function

```
XSIF_ERROR xsif_write_generic(xuint32* num_address, xuint32* address, xuint32 address_step, xuint32 control, xuint32 num_writes, xuint32* data, xuint32* status)
```

### Matlab Function

```
[status, XSIF_ERROR] = xsif_write_generic(address, address_step, control, num_writes, data)
```

### Parameters

| | |
|---|---|
| `num_address` | Number of addresses given in the `address` parameter |
| `address` | An array of addresses that will be written to in sequence. |
| `address_step` | A modifier added to each address in the sequence. |
| `control` | The control word, excluding the write bit. The same control word is used for all writes. The write bit is inserted automatically. |
| `num_writes` | The total number of writes to perform. This must be a multiple of `num_address`. |
| `data` | An array of data words to be written. |
| `status` | An array of returned status words. No status words are returned if `status` is zero. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_write()`

Performs a single SIF write.

This function is provided for convenience and is equivalent to `xsif_write_generic(1, &address, 0, control, 1, data, status)`.

### C Function

```
XSIF_ERROR xsif_write(xuint32 address, xuint32 control, xuint32 data, xuint32* status)
```

### Matlab Function

```
[status, XSIF_ERROR] = xsif_write(address, control, data)
```

### Parameters

| | |
|---|---|
| address | The address to write. |
| control | The control word, excluding the write bit. The write bit is inserted automatically. |
| data | The data word to be written. |
| status | The returned status word. No status word is returned if status is zero. |

### Error Conditions

XSIF_OK if successful.

---

## *function* xsif_write_block()

Performs a set of SIF writes to series of addresses separated by a fixed step.

This function is provided for convenience and is equivalent to xsif_write_generic(1, &address, address_step, control, num_writes, data, status).

### C Function

```
XSIF_ERROR xsif_write_block(xuint32 address, xuint32 address_step,
xuint32 control, xuint32 num_writes, xuint32* data, xuint32* status)
```

### Matlab Function

```
[status, XSIF_ERROR] = xsif_write_block(address, address_step,
control, data)
```

### Parameters

| | |
|---|---|
| address | The starting address. |
| address_step | A modifier added to the address after each write. |
| control | The control word, excluding the write bit. The same control word is used for all writes. The write bit is inserted automatically. |
| num_writes | The total number of writes to perform. |
| data | An array of data words to be written. |
| status | An array of returned status words. No status words are returned if status is zero. |

### Error Conditions

XSIF_OK if successful.

### *function* `xsif_cycle_generic()`

Performs a generic SIF cycle consisting of a number data shifts each with an optional `SIF_LOADB` handshake. This function can be used to implement any SIF access, including the XAP2 abort pending SIF and SIF Command Mode as described in C4862-TM-005.

#### C Function

```
XSIF_ERROR xsif_cycle_generic(xuint32 num_cycles, xuint32* loadb,
xuint32* num_bits, xuint32* mosi, xuint32* miso)
```

#### Matlab Function

```
[miso, XSIF_ERROR] = xsif_cycle_generic(loadb, num_bits, mosi)
```

#### Parameters

| | |
|---|---|
| `num_cycles` | The number of cycles to perform. This is the number of elements in `mosi`, `miso` and `loadb`. |
| `loadb` | An array of flags controlling how `SIF_LOADB` is used after each shift.<br>0 = no handshake<br>1 = handshake<br>2 = ignore SIF_LOADB |
| `num_bits` | An array of shift lengths in bits. The maximum number is 32. |
| `mosi` | An array of output words of bits to be shifted from the master to the slave. |
| `miso` | An array of returned input words of bits shifted from the slave to the master. No input words are returned if `miso` is zero. |

#### Error Conditions

`XSIF_OK` if successful.

### *function* `xsif_cycle()`

Performs a single SIF cycle.

This function is provided for convenience and is equivalent to `xsif_cycle_generic(1, &loadb, &num_bits, &mosi, miso)`.

#### C Function

```
XSIF_ERROR xsif_cycle(xuint32 loadb, xuint32 num_bits, xuint32 mosi,
xuint32* miso)
```

### Matlab Function

```
[miso, XSIF_ERROR] = xsif_cycle(loadb, num_bits, mosi)
```

### Parameters

| | |
|---|---|
| loadb | Controls how SIF_LOADB is used after the shift.<br>0 = no handshake<br>1 = handshake<br>2 = ignore SIF_LOADB |
| num_bits | Number of bits in the shift. The maximum number is 32. |
| mosi | The output word of bits shifted from the master to the slave. |
| miso | The returned input word of bits shifted from slave to master. No input word is returned if miso is zero. |

### Error Conditions

XSIF_OK if successful.

---

## *function* xsif_contread_start()

Starts a continuous read. The continuous read can be thought of as an xsif_read_array() operation where the number of reads is unknown and the data is returned in blocks.

The calling application must use the returned ID, contread_id, in subsequent calls to xsif_contread_get(), xsif_contread_done() and xsif_contread_stop(). The num_reads argument specifies how many SIF reads worth of data and status words will be returned by subsequent calls to xsif_contread_get().

### C Function

```
XSIF_ERROR xsif_contread_start(xuint32 num_address, xuint32* address,
xuint32 control, xuint32 num_reads, xuint32* contread_id)
```

### Matlab Function

```
[contread_id, XSIF_ERROR] = xsif_contread_start(address, control,
num_reads)
```

### Parameters

| | |
|---|---|
| num_address | Number of addresses given in the address parameter. |
| address | An array of addresses that will be read from in sequence. |
| control | The control word, excluding the write bit. The same control |

| | |
|---|---|
| | word is used for all reads. |
| num_reads | The number of SIF Reads worth of data and status to be returned per call to xsif_contread_get(). |
| contread_id | Returned ID. |

**Error Conditions**

XSIF_OK if successful.

---

***function* xsif_contread_get()**

Polls for new data from the continuous read operation.

Pointers to a new blocks of data and status words are returned if new data is available. The number of words in each of these blocks is returned in num_reads. The pointers are valid until xsif_contread_done() or xsif_contread_stop() is called.

If the buffer has overflowed; i.e. this function has not been called often enough, the value at overrun will be set to 1. Otherwise this will be set to 0.

The value returned in num_reads is set to zero if new data is not yet available.

The application must call xsif_contread_done() when it has finished processing the data and status words. Following this call, the application must not access the old data and status words.

Additional calls to xsif_contread_get() are not allowed until xsif_contread_done() is called.

**C Function**

```
XSIF_ERROR xsif_contread_get(xuint32 contread_id, xuint32** data,
xuint32** status, xuint32* num_reads, xuint32* overrun)
```

**Matlab Function**

```
[data, status, overrun, XSIF_ERROR] = xsif_contread_get(contread_id)
```

**Parameters**

| | |
|---|---|
| contread_id | The ID returned from xsif_contread_start(). |
| data | Pointer to an array of new data values. This will be set to 0 if no new data is available. |
| status | Pointer to an array of new status values. This will be set to 0 if no new data is available. |
| num_reads | Number of returned words in data and status. This will be the same as specified in xsif_contread_start() when there are |

| | |
|---|---|
| | no errors. This will be set to zero if no new data is available. |
| `overrun` | Returned value indicating if there has been an overrun.<br>0 = no overrun<br>1 = overrun |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_contread_done()`

Informs the driver that the application has finished processing the last block data and status words. The application must not access the arrays of data and status words from the preceding `xsif_contread_get()` following this call. The `contread_id` argument is the value returned by the call to `xsif_contread_start()` that started the continuous read.

### C Function

```
XSIF_ERROR xsif_contread_done(void)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_contread_done(contread_id)
```

### Parameters

| | |
|---|---|
| `contread_id` | The ID returned from `xsif_contread_start()`. |

### Error Conditions

`XSIF_OK` if successful.

---

## *function* `xsif_contread_stop()`

The `xsif_contread_stop()` function stops a continuous read. The `contread_id` argument is the value returned by the call to `xsif_contread_start()` that started the circular buffer.

### C Function

```
XSIF_ERROR xsif_contread_stop(xuint32* contread_id)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_contread_stop(contread_id)
```

**Parameters**

`contread_id` The ID returned from `xsif_contread_start()`.

**Error Conditions**

`XSIF_OK` if successful.

## 4.6 xSIF IO Functions

---

***function* `xsif_claim()`**

Attempts to claim the SIF for the current slave. For single master SIF, the SIF signals are enabled when `xsif_claim()` is called.

If there are multiple masters, arbitration is performed by asserting `SIF_REQ` and waiting for `SIF_ACK` to be asserted. When waiting for `SIF_ACK`, there is a timeout equal to the value specified in the `ack_timeout` element of the `sif_slave_config` structure. If chip selects are being used, the chip select of the current slave is enabled and asserted. When the SIF is claimed the SIF signals and the chip select of the current slave tri-states are enabled.

**C Function**

```
XSIF_ERROR xsif_claim(void)
```

**Matlab Function**

```
[XSIF_ERROR] = xsif_claim()
```

**Error Conditions**

`XSIF_OK` if successful.

---

***function* `xsif_release()`**

Releases the SIF for the current slave. For single master system, the SIF signals are disabled when `xsif_release()` is called.

If there are multiple masters `SIF_REQ` signal is released and the function waits for `SIF_ACK` to be released. When waiting for `SIF_ACK`, there is a timeout equal to the value specified in the `ack_timeout` element of the `sif_slave_config` structure. The signals associated with SIF and the chip select of the current slave are disabled.

**C Function**

```
XSIF_ERROR xsif_release(void)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_release()
```

### Error Conditions

XSIF_OK if successful.

---

## *function* xsif_get_io_dir()

Returns the direction of one of the IO signals.

### C Function

```
XSIF_ERROR xsif_get_io_dir(XSIF_SIGNAL signal, xuint32* direction)
```

### Matlab Function

```
[direction, XSIF_ERROR] = xsif_get_io_dir(signal)
```

### Parameters

| | |
|---|---|
| signal | Identifies an IO signal. |
| direction | The direction of the signal.<br>0 = output<br>1 = input |

### Error Conditions

XSIF_OK if successful.

---

## *function* xsif_set_io_dir()

Configures the direction of one of the IO signals.

The XSIF_SIGNAL_ENABLE_ALL constant refers to the signal that is used to globally enable and disable all signals from the SIF Pod. This signal is always an output and cannot be altered by using the xsif_set_io_dir function.

### C Function

```
XSIF_ERROR xsif_set_io_dir(XSIF_SIGNAL signal, xuint32 direction)
```

### Matlab Function

```
[XSIF_ERROR] = xsif_set_io_dir(signal, direction)
```

### Parameters

| | |
|---|---|
| signal | Identifies an IO signal. |

| | |
|---|---|
| direction | The direction of the signal.<br>0 = output<br>1 = input |

**Error Conditions**

XSIF_OK if successful.

---

### *function* xsif_get_io()

The xsif_get_io() function allows the state of any of the signals to be read at any time.

**C Function**

```
XSIF_ERROR xsif_get_io(XSIF_SIGNAL signal, xuint32* value)
```

**Matlab Function**

```
[value, XSIF_ERROR] = xsif_get_io(signal)
```

**Parameters**

| | |
|---|---|
| signal | Identifies an IO signal. |
| value | The input value.<br>0 = low<br>1 = high |

**Error Conditions**

XSIF_OK if successful.

---

### *function* xsif_set_io()

Sets the output state of one of the IO signals. This function can be used for the general IO signals and the internal signals used to control the XAP signals. It is possible to change the output state of a signal configured as an input. The new state will take effect when the signal is next configured as an output.

**C Function**

```
XSIF_ERROR xsif_set_io(XSIF_SIGNAL signal, xuint32 value)
```

**Matlab Function**

```
[XSIF_ERROR] = xsif_set_io(signal, value)
```

### Parameters

| | |
|---|---|
| `signal` | Identifies an IO signal. |
| `value` | The output value.<br>0 = low<br>1 = high |

### Error Conditions

`XSIF_OK` if successful.

Claims

1. A system comprising:
a master computer;
one or more control pods; and
one or more integrated circuits, each comprising one or more processors;
wherein the master computer is operable to interact with any of said processors via said one or more control pods.

2. A system according to claim 1, wherein the or each integrated circuit comprises an interface circuit which is operable i) to provide an interface for a control pod to signal a request for access to storage locations of the processor; and ii) to provide parity checking for said access which provides odd parity over an even number of bits.

3. A system according to claim 1 or 2, wherein said master computer is operable to interact with any of said processors by issuing a reset instruction.

4. A system according to claim 3, wherein said reset instruction is operable to reset a portion of a selected one of the integrated circuits.

5. A system according to claim 4, wherein said portion is one or more of: a selected processor; an interface circuit that provides access to the processor; off-chip circuitry; and on-chip circuitry.

6. A system according to claim 1 or 2, wherein said master computer is operable to interact with any of said processors by issuing a cancel instruction of a previous instruction issued by the master computer or issued by circuitry internal to said integrated chip.

7. A system according to any of claims 1 to 6, wherein said master computer is operable to request data from or to send data to one or more of said processors and wherein said one or more processors are operable to output a time stamp indicating when the data is read or written to or from memory in or connected to the integrated circuit.

8. A system according to any of claims 1 to 7, wherein said master computer is operable to transmit a query command to said integrated circuit and wherein said integrated circuit is operable to output a response identifying register sizes and processor numbers in said integrated circuit.

9. A system according to any of claims 1 to 8, wherein said control pod is operable to receive plural data segments from said interface circuit and is operable to collate the data fragments and to send them to said master computer in a single data packet.

10. A system according to any of claims 1 to 9, wherein said control pod is operable: i) to receive data from said master computer; ii) to fragment the data into a plurality of smaller data fragments; and iii) to transmit said smaller data fragments one at a time to an integrated circuit.

11. A data processing apparatus comprising:

a plurality of processors each constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set; and

an interface circuit provided in common to said plurality of processors and which is operable to provide an interface for an external apparatus to signal a request for access to a storage location of one of the processors selected by the request.

12. A data processing apparatus according to claim 11, wherein said request includes:

i) an address field for specifying an address;

ii) a control field for selecting the processor; and

iii) a data field for the input and output of data;

wherein said interface circuit includes control circuitry for controlling access to the storage location of the selected processor using said address, control and data fields of said request.

13. An apparatus according to claim 12, wherein said request includes a status field and wherein said interface circuit is operable to use said status field for reporting status information to said external apparatus.

14. An apparatus according to claim 11 or 12, comprising a controller operable, in response to interrogation from an external apparatus, to output data to said external apparatus identifying register lengths of said apparatus and identifying the number of processors coupled in common to said interface circuit.

15. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

master circuitry operable to request access to storage locations of the processor; and

an interface circuit which is operable i) to provide an interface for an external apparatus to signal a request for access to storage locations of the processor; and ii) to provide an interface for said master circuitry to signal a request for access to storage locations of the processor.

16. An apparatus according to claim 15, wherein said master circuitry comprises a processor or a hardware state machine.

17. An apparatus according to claim 15 or 16, wherein a parallel interface is provided between said master circuitry and said interface circuit and a serial interface is provided for connecting the interface circuit to said external apparatus.

18. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

an interface circuit which is operable to provide an interface for an external apparatus to signal a request for access to storage locations of the processor; and

a controller operable to control input and output to and from the interface circuit and operable to generate odd parity over an even number of bits to allow the identification by said external apparatus if an input or an output of said interface circuit is stuck at a logic low or a logic high signal.

19. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set; and

an interface circuit which is operable to provide an interface for an external apparatus to signal a request to control the data processing apparatus;

wherein said request to control the data processing apparatus is operable to cause at least a portion of the data processing apparatus to reset.

20. An apparatus according to claim 19, wherein said request is operable to cause said interface circuit to reset.

21. An apparatus according to claim 19, wherein said request is operable to cause said processor to reset.

22. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

an interface circuit which is operable to provide an interface for an external apparatus to signal a request to control the data processing apparatus; and

one or more counters for maintaining counts relating to the operation of the interface circuit;

wherein in response to receiving a request for a count held by at least one of said one or more counters, said interface circuit is operable to read the value of said at least one counter and to output the value to said external apparatus.

23. An apparatus according to claim 22, wherein counters are provided to count one or more of: the number of data reads requested by the external apparatus; the number of data writes requested by the external apparatus; the number of errors; and the number of cancelled requests that have been made by the external apparatus.

24. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

an interface circuit which is operable to provide an interface for an external apparatus to signal a request to control the data processing apparatus;

wherein pending requests are stored in said interface circuit; and

a control circuit operable to cancel a pending request upon receipt of a cancel request from said external apparatus.

25. An apparatus according to claim 24, further comprising master circuitry operable to request access to storage locations of said processor via said interface circuit and wherein said control circuit is operable to cancel a request from said master circuitry upon receipt of said cancel request from said external apparatus.

26. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

an interface circuit which is operable to provide an interface for an external apparatus to signal a request for data from a storage location of the processor;

a controller operable to cause the data to be read from the storage location, operable to generate a time stamp indicating a time at which the data was read from said storage location and operable to cause said read data and said time stamp to be output to said external device.

27. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

an interface circuit which is operable to provide an interface for an external apparatus to signal a request for access to storage locations of the processor; and

acquisition circuitry operable to acquire signals from an anaologue input;

a controller operable to prevent output of signals to said external apparatus at a time that said acquisition circuitry is acquiring said signals from said analogue input.

28. An apparatus according to claim 27, wherein said controller comprises a hardware circuit which is operable to enable and disable output of signals from said interface circuit to said external apparatus.

29. An apparatus according to claim 27, wherein said controller is defined by one or more of said instructions that control the operation of said processor, such that upon execution of a specific instruction allows said interface circuit to provide said external apparatus with access to said storage location.

30. A data processing apparatus comprising:

a processor constructed to operate under control of a stored program comprising a sequence of program instructions selected from a predetermined instruction set;

an interface circuit which is operable to provide an interface for an external apparatus to signal a request for access to storage locations of the processor;

wherein said instructions include a number of instructions that allow said interface circuit to provide said external apparatus with said access; and

wherein one or more of said instructions that allow said interface circuit to provide said external apparatus with said access is defined by all ones or all zeros.

31. An apparatus according to any of claims 11 to 30, comprising control means operable to provide access between a specified storage location and the interface circuit in response to such a request only at predetermined points in the execution of the stored program by said processor, said control means being operable to fix periods of time for providing such access

relative to the sequence of program instruction such that execution timing of the stored instructions is independent of whether or not such a communication request is supplied to said circuit by said external apparatus.

32. An apparatus according to claim 31, wherein said control means comprises a generic communication instruction of the instruction set, the generic communication instruction having a fixed execution time by said processor.

33. An apparatus according to any of claims 11 to 32, wherein said processor and interface circuit are provided on a single integrated circuit.

34. An apparatus according to any of claims 11 to 33, wherein said interface circuit provides a serial interface to said external apparatus.

35. An apparatus according to claim 34, wherein said serial interface includes separate data lines for input and output of data to the interface circuit.

36. An apparatus according to claim 34 or 35, wherein said serial interface includes a clock line for synchronising said data input and output asynchronously with respect to operation of said processor.

37. An apparatus according to any of claims 11 to 36, wherein the communication request includes a specific communication instruction loaded by the external apparatus into an interface register of the interface circuit, and wherein the control means is operable to provide access between the specified storage location and the interface register under control of the specific communication instruction.

38. An apparatus according to claim 37, wherein the interface circuit includes at least one control line which has an active state which is setable by either of the interface circuit and the external apparatus, wherein said external apparatus is operable to signal said request for access by setting the active state of said at least one control line, wherein said interface circuit is operable to detect the setting of said active state of said at least one control line by said external apparatus and to maintain the active state on said at least one control line until access has been provided between the specified storage location and the interface circuit during one of said periods of time and wherein said interface circuit is operable to release said control line when it is ready to receive a subsequent communication request.

39. An apparatus according to claim 38, wherein, at least for a specific communication instruction which is recognised by the processor as requiring information in response, said processor is operable to load said information into the interface register of the interface circuit during said time period, before the interface circuit releases the control line, which allows said information to be read from the interface register by the external apparatus.

40. An apparatus according to any of claims 37 to 39, comprising circuitry which is operable to interpret the specific communication instructions in accordance with pre-determined rules.

41. An apparatus according to any of claims 37 to 40, wherein said specific communication instruction includes an address field which is interpreted by the processor as specifying said storage location within a storage space of the processor, and wherein the control means is responsive to the specific communication instruction during said time period to allow reading, writing or a selection of reading and writing to be performed between the interface circuit and the specified storage location.

42. An apparatus according to claim 41, wherein said storage space comprises data memory space, register space, and/or program memory of the processor.

43. An apparatus according to claim 41 or 42, wherein said external apparatus is operable to select a reading or writing operation within the processor under control of a selection field in the specific communication instruction loaded into the interface register.

44. An apparatus according to any of claims 41 to 43, wherein said interface register is a shift register which is loadable bit-serially under control of said external apparatus.

45. An apparatus according to any of claims 11 to 44, wherein the processor is responsive to a predetermined external signal so as to enter a state in which program controlled operation of the processor is suspended, and wherein said control means is responsive to received communication requests to provide said access at any time during said suspended state.

46. An apparatus according to any of claims 11 to 45, further comprising mixed analogue and digital signal processing circuitry integrated with said processor in a single integrated circuit.

47. A method characterised by the step of using a system according to any of claims 1 to 10.

48. A method characterised by the step of using an apparatus according to any of claims 11 to 46.

49. A processor which comprises an instruction set for execution on the processor, a processor architecture and a memory, wherein at least some of the instructions in the instruction set are adapted to write to and/or read from any adjacent memory register pair.

50. A processor according to Claim 49, wherein the instructions are able to write to and/or read from unaligned register pairs.

51. A processor according to Claim 49, wherein the instructions are able to write to and/or read from aligned register pairs.

52. A processor according to any of the preceding claims 49 to 51, wherein the instructions are able to write to and/or read from any adjacent register pair within a single program instruction cycle.

The Patent Office

Innovation

**pplication No:** GB0524772.1 **Examiner:** Mr David Maskery

**laims searched:** 1 - 10 and 47 **Date of search:** 27 July 2006

## atents Act 1977: Search Report under Section 17

**ocuments considered to be relevant:**

| ategory | Relevant to claims | Identity of document and passage or figure of particular relevance |
|---|---|---|
| X,Y | X 1, 2, 6 - 10, 47 and Y 3 - 5 | GB 2156550 A (DAISY SYSTEMS) See pages 3 - 5 and fig 1. |
| X,Y | X 1, 2, 6 - 10, 47 and Y 3 - 5 | GB 2217059 A (BENCHMARK TECH) See pages 45 - 49, 61 and figs. |
| X,Y | X 1, 2, 6 - 10, 47 and Y 3 - 5 | GB 2329984 A (THOMSON TRAINING & SIMULATION) See pages 4 - 6 and fig 1. |
| Y | 3 - 5 | US 5815144 B (IBM) See column 2 and fig 3. |
| Y | 3 - 5 | US 5870602 B (COMPAQ) See columns 3, 7 - 11 and fig 6. |

egories:

| | | |
|---|---|---|
| Document indicating lack of novelty or inventive step | A | Document indicating technological background and/or state of the art. |
| Document indicating lack of inventive step if combined with one or more other documents of same category. | P | Document published on or after the declared priority date but before the filing date of this invention. |
| Member of the same patent family | E | Patent document published on or after, but with priority date earlier than, the filing date of this application. |

**ld of Search:**

rch of GB, EP, WO & US patent documents classified in the following areas of the UKC$^{X}$ :

A

rldwide search of patent documents classified in the following areas of the IPC

6F

following online and other databases have been used in the preparation of this search report

ODOC, WPI.