STMicroelectronics

ST231 core and instruction set architecture

Reference manual

7645929 Rev N

September 2009

BLANK

Reference manual

ST231 core and instruction set architecture

Introduction

The 32-bit ST231 is a member of the ST200 family of cores.

This family of embedded processors uses a scalable technology that allows variation in instruction issue width, the number and capabilities of functional units and register files, and the instruction set.

The ST200 family includes the following features:

- parallel execution units, including multiple integer ALUs and multipliers
- architectural support for data prefetch
- predicated execution through select operations
- efficient branch architecture with multiple condition registers
- encoding of immediate operands up to 32 bits
- support for user and supervisor modes and memory protection

Contents

Preface

ST200 document identification and control

Each book in the ST200 documentation suite carries a unique ADCS identifier of the form:

ADCS *nnnnnnnx*

where *nnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on an ST200 document, the complete identification ADCS *nnnnnnnx* should be quoted.

ST200 documentation suite

The ST200 documentation suite comprises the following volumes:

ST231 Core and Instruction Set Architecture

(ADCS 7645929) This manual describes the architecture and the instruction set of the ST231 core as used by STMicroelectronics.

ST200 User Manual

(ADCS 8063762) This manual describes the ST200 Micro Toolset and provides an introduction to OS21. It covers the various cross tools and libraries that are provided in the toolset, the target platform libraries, how to boot OS21 applications from ROM and how to port applications which use STMicroelectronics' OS20 operating systems. Information is also given on how to build the open source packages that provide the compiler tools, base run-time libraries and debug tools and how to set up an ST Micro Connect.

ST200 Micro Toolset Compiler Manual

(ADCS 7508723) This manual provides a detailed guide to using the ANSI C and C++ compiler drivers for compiling and linking source code to produce an executable binary. The compiler drivers are introduced in terms of how they fit into the complete ST200 toolchain. The manual then concentrates on the facilities provided by the compiler drivers to produce efficient code. It covers: command line options, predefined macros, supported pragmas, compiler optimization techniques, GNU C and C++ language extensions and `asm` construct, the assembly language and intrinsic functions.

ST200 Run-time Architecture Manual

(ADCS 7521848) This manual describes the common software conventions for the ST200 processor run-time architecture.

ST200 ELF Specification

(ADCS 7932400) This document describes the use of the ELF file format for the ST200 processor. It provides information needed to create and interpret ELF files and is specific to the ST200 processor.

OS21 User Manual

*(*ADCS 7358306) This manual describes the royalty free, light weight, OS21 multitasking operating system.

OS21 for ST200 User Manual

(ADCS 7410372) This manual describes the use of OS21 on ST200 platforms. It describes how specific ST200 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST200 platforms.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `sample code`, `keyboard input` and `file names`
- *variables* and *`code variables`*
- *`code comments`*,
- **screens**, **windows** and **dialog boxes**
- **instructions**

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES
- PIN NAMES and SIGNAL NAMES

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

- Terminal strings of the language, that is, strings not built up by rules of the language, are printed in teletype font. For example, `void`.
- Nonterminal strings of the language, that is, strings built up by rules of the language, are printed in italic teletype font. For example, *`name`*.
- If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-`*`name`*.
- Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’).
- Alternatives are separated by vertical bars (‘`|`’).
- Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’).
- Items which may be repeated appear in braces (‘`{`’ and ‘`}`’).

Acknowledgements

The ST231 core is based on technology jointly developed by Hewlett-Packard Laboratories and STMicroelectronics.

1 Overview

This chapter provides an introduction to the ST231 processor and to this reference manual.

1.1 VLIW overview

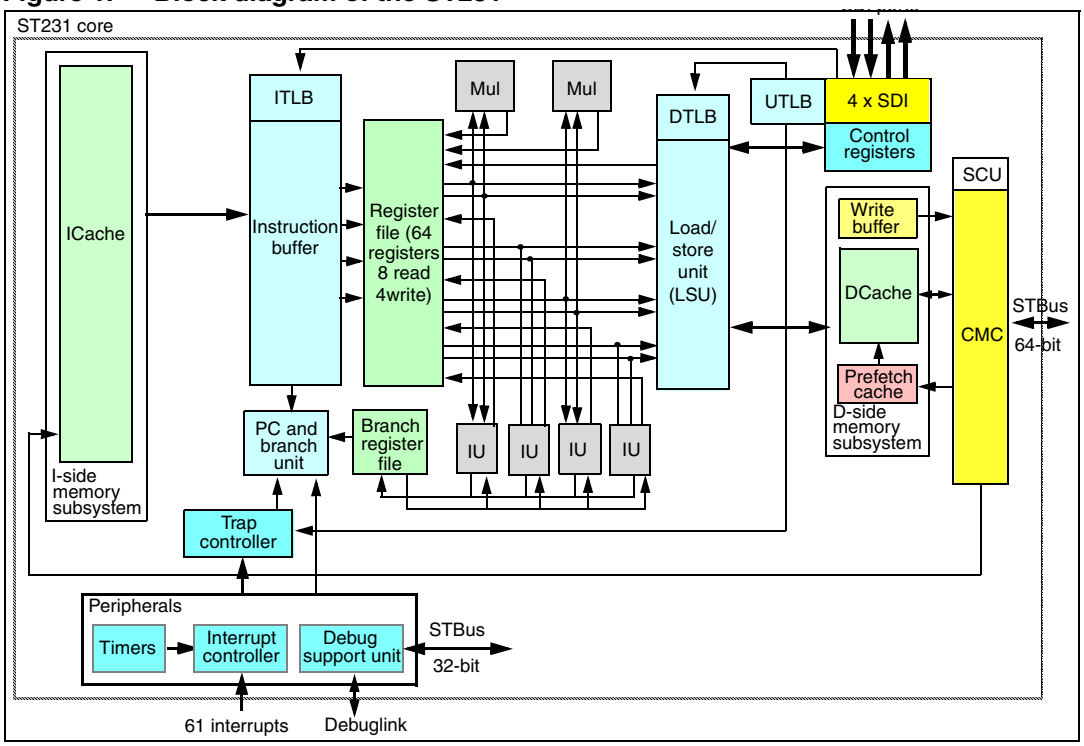
VLIW (very long instruction word) processors use a technique where instruction level parallelism is explicitly exposed to the compiler, which must schedule operations to account for the operation latency. The hardware implementation of a VLIW processor is significantly simpler than a corresponding multiple issue superscalar CPU because of the simplicity of the grouping and scheduling hardware; the complexity is passed to the instruction scheduling software (compiler and assembler) which is responsible for scheduling the parallel operations for maximum efficiency.

RISC-like operations (syllables) are grouped into bundles (wide words). The operations in a bundle are issued simultaneously. In the ST200 family operations also complete simultaneously. While the delay between issue and completion is the same for all operations, some results are available for bypassing to subsequent operations prior to completion. This is discussed further in *Chapter 4: Execution pipeline and latencies on page 22*.

1.2 ST231 overview

The ST231 includes the ST231 core and associated peripherals. *Figure 1* shows the arrangement of these components in a block diagram.

Figure 1. Block diagram of the ST231



1.3 Document overview

This manual describes the architecture and instruction set of the ST231 implementation. This section gives an outline of the following document.

The processor is made up of a number of functional units described in *Chapter 2: Execution units* which operate on data stored in the register files (*Chapter 3: Architectural state*). These functional units are pipelined and subject to explicit observable latencies (*Chapter 4: Execution pipeline and latencies*).

The handling of exceptions and interrupts are detailed in *Chapter 5: Traps (exceptions and interrupts)*.

The ST231 accesses memory through the memory subsystem (*Chapter 7: Memory subsystem*) which provides protection and address translation by means of a translation lookaside buffer (*Chapter 5: Traps (exceptions and interrupts)*).

The ST231 has four SDI ports (*Chapter 8: Streaming data interface*) which allow it to communicate rapidly with other devices and avoid cache pollution when processing large amounts of data.

Control of the devices is performed using the memory mapped control registers defined within the relevant chapters. The address of the control registers and PSW are detailed in *Chapter 9: Control registers*.

The ST231 also provides a performance monitoring system to help with software optimization and debugging (*Chapter 15: Performance monitoring*).

The following peripheral devices are also provided: timers (*Chapter 10: Timers*), interrupt control (*Chapter 12: Interrupt controller*) and debug support[a] (*Chapter 13: Debugging support (TAPLink)* or *Chapter 14: Debugging support (JTAG)*). The peripheral register addresses are detailed in *Chapter 11: Peripheral addresses*.

The execution model is described in *Chapter 16: Execution model*. The execution of bundles is described in *Section 16.1: Bundle fetch, decode, and execute on page 116*, including the behavior of the machine when exceptions or interrupts are encountered.

Chapter 18: Instruction set describes the details of each operation, including the semantics. The instruction set includes details of the instruction set encoding, syntax and semantics. The encoding of bundles is defined in *Section 18.1: Bundle encoding on page 137*.

The behavior of operations is specified using the notational language defined in *Chapter 17: Specification notation on page 119* through *Section 17.3: Statements on page 125*. The descriptions clearly identify where architectural state is updated and the latency of the operands.

A simple model of memory and control registers defined in *Section 17.5.2: Memory model on page 130* and *Section 17.5.3: Control register model on page 134* is used when specifying load and store operations.

a. Only one of the debugging support chapters is applicable depending upon the version of the core implemented. Please refer to the product datasheet for specific variant details.

2 Execution units

The functional core of ST231 comprises of a number of execution units working on two register files. The execution units include 4 integer units, 2 multiply units, a load/store unit and a branch unit which are all described in this chapter. The two register files, the branch registers and the general purpose registers are described in *Chapter 3: Architectural state on page 19*.

2.1 Integer units (IU)

The ST231 has four identical integer units. Each integer unit is capable of executing one operation per cycle. The results of the integer units can be used as operands of the next bundle. This is equivalent to a pipeline depth of one cycle.

Each operation can take up to three operands in the form of two 32-bit values and a single conditional bit. The IU then executes the appropriate operation and produces up to two results in the form of a 32-bit value and a 1-bit conditional value. The integer operations supported are detailed in the *Chapter 18: Instruction set on page 137*.

2.2 Multiply units

The ST231 has two identical multiply units. Each multiply unit is pipelined with a depth of three cycles, executing an operation every cycle.

Each multiply units takes two 32-bit operands and produces a single 32-bit result. The multiply operations supported are detailed in the *Chapter 18: Instruction set on page 137*.

2.3 Load/store unit (LSU)

The ST231 has a single load/store unit. The load/store unit is pipelined with a depth of three cycles, executing an operation every cycle.

The load store can take up to three 32-bit operands and may produce a single 32-bit result depending on the operation. The load store operations supported are detailed in the *Chapter 18: Instruction set on page 137*.

Memory access protection and translation is implemented by the TLB, this is part of the memory sub-system. The TLB also controls the cache behavior of data accesses, *Chapter 6: Memory translation and protection on page 31*.

Uncached accesses or accesses which miss the data cache cause the load/store unit to stall the pipeline to ensure correct operation.

2.3.1 Memory access

The ST231 uses a single 32-bit address space to address the external memory system. Peripheral devices and control registers are also mapped into the address space.

All cacheable memory transactions are made using the data cache. The data cache determines if an external memory access (using the STBus) is required to complete the request.

*Note: Cacheable **STORE** memory transactions that miss are written to the write buffer not the data cache.*

Uncached accesses are performed directly on the memory system, see to *Section 7.3.5: Uncached load and stores on page 53*.

2.3.2 Addressing modes

The ST231 supports one addressing mode – the effective address is an immediate (constant) plus a register.

2.3.3 Alignment

All **LOAD** and **STORE** instructions work on data stored on the natural alignment of the data type; that is, words on word boundaries, half-word on half word boundaries.

LOAD and **STORE** operations with misaligned addresses raise an exception which makes possible the implementation of misaligned **LOAD**s by trap handlers.

For a byte or half-word **LOAD**, the data from memory is loaded into the least significant part of a register and is either sign-extended or zero extended according to the instruction definition.

For a byte or half-word **STORE**, the data stored from the least significant part of a register.

2.3.4 Control registers

The LSU maps a part of the address space that is devoted to control registers (see *Chapter 9: Control registers on page 67* for details). The LSU control register block intercepts **LOAD**s and **STORE**s to this area of memory so that it can process the operation. No access to the data cache is made for control register operations. Transactions are made across the 32-bit control register bus to those control registers that live outside the LSU.

2.3.5 Cache purging

Cache purging (flush and invalidate) operations are provided on the ST231.

They allow for purging lines and sets from the data cache, and invalidating the entire instruction cache.

2.3.6 Dismissible loads

Dismissible **LOAD**s are used to support software load speculation. This allows the compiler to schedule a **LOAD** in advance of a condition that predicates its use.

Dismissible **LOAD**s are required to return the same value as a normal **LOAD** if such an operation can be executed without causing an exception. Otherwise dismissible **LOAD**s return zero.

In the event that misaligned accesses are supported through a software trap handler, the ST231 may be configured to trap non-aligned dismissible LOADs, see the *Chapter 5: Traps (exceptions and interrupts) on page 25*. The TLB can be configured to return zero for dismissible LOADs in cases where they can be executed without exception; this is to support peripherals which have destructive read behavior.

2.4 Branch unit

The ST231 has one branch unit. This unit supports both relative immediate branches (with and without condition code) and absolute and relative jumps and calls.

A conditional branch is performed using the **BR** and **BRF** instructions. These instructions have two operands, a condition code register and the immediate offset of the branch.

An unconditional branch is performed using the **GOTO** (immediate) instruction. This instruction has one operand containing the immediate offset of the branch.

An unconditional jump is performed using the **GOTO** (link register) instruction. This instruction causes a control transfer to the address stored in the link register, see *Section 3.2.1: Link register on page 19*.

An unconditional call is performed using the **CALL** (link register) and **CALL** (immediate) instructions. These instruction cause a control transfer to the address stored in the link register (see *Section 3.2.1: Link register on page 19*) or to the specified immediate offset. After the call, the link register contains the return address.

Due to pipeline restrictions all branches and jumps incur a penalty of one cycle of stall.

2.4.1 Idle mode macro

The **IDLE mode** macro is encoded as a bundle containing a **GOTO** (immediate) to the same bundle. The **IDLE mode** macro must be alone in a bundle (otherwise it is treated as a normal **GOTO**).

The **IDLE mode** macro is architecturally identical to the branch it is derived from. When an interrupt or debug interrupt occurs the core exits idle mode and jumps to the correct handler.

Implementation notes

When an **IDLE mode** macro is executed the ST231:

- empties the pipeline, completing any instructions issued before the idle,
- waits for all outstanding bus transactions to complete:
  - all prefetches issued to the bus have completed (responses have come back)
  - all writes issued to the bus have completed (responses have come back)
- waits for the SDI output buffer to be become empty
- enters idle mode

The core will not enter idle mode while the performance monitoring hardware is enabled. When the core enters idle mode a bit is set in the PM_CR register, see *Section 15.3: Control register (PM_CR) on page 114*.

The core discontinues entry to idle mode and jumps to the correct handler on the following conditions:

- STBus error exception
- external interrupt
- debug interrupt

The core exits idle mode and jumps to the correct handler on the following conditions:

- external interrupt
- debug interrupt

While in idle mode:

- timers continue to operate normally
- the SDI input ports do not accept data
- the SDI output ports do not send out data as they must be empty before the core enters idle mode
- the peripheral blocks accepts and responds to STBus transactions as normal
- the DSU continues to operate as normal (both using the TAPLink and using the STBus)

2.4.2 syncins macro

The **syncins** macro can be used to ensure that all previous instructions have completed and all new instructions have not yet started. The **syncins** macro ensures that the pipeline is empty and the instruction buffer is purged.

The **syncins** macro may only be executed in supervisor mode.

3 Architectural state

This chapter describes the architectural state of the ST231 core, which consists of the following elements:

- program counter
- register file
- branch register file
- program status word
- control registers

3.1 Program counter (PC)

The PC contains a 32-bit byte address pointing to the beginning of the current bundle in memory. The two LSBs of the PC are always zero.

3.2 Register file

The general purpose register file contains 64 words of 32 bits each. These are named R0 to R63.

Reading register zero (R0) always returns the value zero. Writing values to R0 has no effect on the processor state.

3.2.1 Link register

Register R63, the architectural link register, is used by the **call** and **return** mechanism. R63 is updated by explicit register writes and the **call** operation. Some restrictions apply to accessing the link register, see *Section 4.5.2: Restrictions on link register on page 24*.

3.3 Branch register file

The branch register file contains eight single bit branch registers, B0 to B7.

3.4 Program status word (PSW)

The program status word (PSW) contains control information that affects the operation of the ST231 processor.

3.4.1 Bit fields

The PSW contains the bit fields listed in *Table 1*.

Table 1. PSW bit fields

Name	Bit(s)	Writable	Reset	Comment
USER_MODE	0	RW	0x0	0: the core is in supervisor mode 1: the core is in user mode
INT_ENABLE	1	RW	0x0	0: external interrupts are disabled 1: external interrupts are enabled
TLB_ENABLE	2	RW	0x0	0: address translation is disabled 1: address translation is enabled
TLB_DYNAMIC	3	RW	0x0	0: speculative loads and purge address ignore “no mapping” violations. 1: speculative loads and purge address cause “no mapping” violations.
SPECLOAD_MALIGNTRAP_EN	4	RW	0x0	0: disables exceptions on speculative load misalignment errors. 1: enables exceptions on speculative load misalignment errors.
Reserved	5	RO	0x0	Reserved
Reserved	6	RO	0x0	Reserved
Reserved	7	RO	0x0	Reserved
DBREAK_ENABLE	8	RW	0x0	0: data breakpoints are disabled 1: data breakpoints are enabled
IBREAK_ENABLE	9	RW	0x0	0: instruction breakpoints are disabled 1: instruction breakpoints are enabled
Reserved	10	RO	0x0	Reserved
Reserved	11	RO	0x0	Reserved
DEBUG_MODE	12	RW	0x0	0: the core is not in debug mode 1: the core is in debug mode
Reserved	[31:13]	RO	0x0	Reserved

3.4.2 USER_MODE

The USER_MODE bit indicates whether the machine is in user mode or supervisor mode. When in user mode, the processor has restricted access:

- the TLB (see *Chapter 6: Memory translation and protection on page 31*) defines the level of access to memory in both user and supervisor modes
- in user mode there is limited access to control registers, see *Chapter 9: Control registers on page 67*
- certain instructions can not be executed in user mode, see *Chapter 18: Instruction set on page 137*

3.4.3 DEBUG_MODE

The DEBUG_MODE bit indicates whether the machine is in debug mode. For the effect of writing to DEBUG_MODE, see *Exiting debug mode on page 88*.

3.4.4 PSW access

The PSW can be read as a control register, *Section 3.5: Control registers on page 21*.

The **pswset** instruction is used to set any number of bits in the PSW atomically. The **pswclr** instruction is used to clear any number of bits in the PSW atomically.

The PSW can also be updated by means of an **rfi** operation. The required status word should be stored into the SAVED_PSW and the address of the code to be executed directly after the change should be stored in the SAVED_PC. Then executing an **rfi** atomically copies the SAVED_PSW into the PSW and the SAVED_PC into the PC.

Example: Procedure to write the PSW, (in ST231 assembler code),

```
_sys_set_psw
   stw SAVED_PC[$r0] = $r63;; // Return address
   stw SAVED_PSW[$r0] = $r4;; // New value
   nop ;;
   nop ;;
   nop ;;
   nop ;;
   rfi ;;
```

Note: Interrupts must be disabled during this sequence to prevent SAVED_PC and SAVED_PSW from being changed.

3.5 Control registers

Additional architectural state is held in a number of memory mapped control registers, *Chapter 9: Control registers on page 67*. These registers include support for interrupts and exceptions, and memory protection.

4 Execution pipeline and latencies

This chapter describes the architecturally visible pipeline and operation latencies for the ST231.

4.1 Execution pipeline

The ST231 uses a pipelined execution scheme. This pipeline is architecturally visible in a number of areas:

- operation latencies
- branch stalls
- bypassing
- usage restrictions

The execution pipeline is three cycles long and comprises three stages E1, E2 and E3. All operations begin in E1. Operands are read or bypassed to an operation at the start of E1. All results are written at the end of E3.

This execution pipeline allows arithmetic and **load**/**store** operations to execute for up to three cycles. The results of operations which complete earlier than E3 are made available for bypassing as operands to subsequent operations, though strictly operations do not complete until the end of the E3 stage. This is when the architectural state is updated.

The pipeline is designed to efficiently implement the serial execution of the code, see *Chapter 16: Execution model on page 116*.

4.2 Operation latencies

ST231 operations begin in E1 cycle and complete in either E1, E2 or E3. The time taken for an operation to produce a result is called the operation latency. For simple operations like **add** and **subtract** the latency is a single cycle. For operations like **multiply** and **load** the latency is three cycles.

Note: Operational latencies may vary between different members of the ST200 processor family.

4.3 Branch stalls

The ST231 has no penalty for not taken branches.

The ST231 stalls for one cycle when a branch is taken. There may be a further stall caused by the destination bundle of a branch crossing an I-cache line boundary. See *Section 7.2.1: Instruction buffer on page 49*.

4.4 Interlocks

The ST231 provides operation latency interlock checking. This enforces the latency between all operations by stalling the pipeline, with the following exceptions:

- store to SAVED_PSW to **rfi**
- store to SAVED_PC to **rfi**
- store to SAVED_SAVED_PSW to **rfi**
- store to SAVED_SAVED_PC to **rfi**

In the cases listed above, the software must ensure that the control register has been updated before executing the **rfi**. See *Section 3.4.4: PSW access on page 21* for further details of how to do this.

For all other cases, the ST231 automatically stalls the pipeline to uphold the internal latency constraints. As such there are no possible latency violations for the above cases.

For optimal machine usage, bundles containing useful operations should be inserted in order to respect the underlying latency between operations.

4.5 Additional notes

Additional information about flushing the pipeline and restrictions on the link register are provided in this section.

4.5.1 Flushing the pipeline

As state is stored within the pipeline, some changes require that state to be flushed out to ensure coherency. For example, the ST231 pipeline needs to be flushed to ensure that UTLB updates take effect. (For the recommended sequence for UTLB updates, see *Coherency on page 40*).

The following instructions cause the pipeline to be emptied:

- **rfi**
- **pswset**
- **pswclr**
- **prgins**
- **prginspg**

4.5.2 Restrictions on link register

To optimize performance, the ST231 contains a speculative link register (SLR). This is a copy of possible future updates to R63. In the current implementation, this register is updated earlier in the pipeline than R63. The core uses SLR as the source for register indirect branch operations.

There are circumstances when SLR is not a true copy of R63. This occurs when an interrupt or exception is taken immediately before an update to R63 but after the SLR has been speculatively changed. The solution to this is to ensure that all interrupt and exception handlers write explicitly to R63 prior to the execution of an **rfi**, **call $r63** or **goto $r63**. This requirement can easily be met with a **mov** operation from R63 to R63 in one of the first bundles of the trap handler.

Register indirect **call** and **goto** operations also require R63 to be stable. If R63 is modified in the three bundles preceding one of these operations, an interlock stall occurs.

A number of operations cannot target R63 for efficiency reasons. These include **multiply** operations, byte and half-word **load** operations, see *Chapter 18: Instruction set on page 137*.

5 Traps (exceptions and interrupts)

In the ST231 architecture, exceptions and interrupts are jointly termed traps. This chapter describes the trap mechanism.

5.1 Trap mechanism

The ST231 defines two types of traps:

- external asynchronous traps (interrupts and bus errors),
- internal synchronous traps (exceptions resulting from operation execution).

A trap point is the point in the program execution where a trap occurs. All bundles executed before the trap point have finished updating the architectural state; no architectural state has been updated by subsequent bundles. For an exception, the trap point is the (start of the) bundle which caused the exception. For an interrupt, the trap point is (the start of) the bundle whose execution has been interrupted. Typically this is a bundle that had been executed shortly after the interrupt was raised or enabled.

The flow diagram, *Figure 16 on page 101* defines when a trap is taken. The aim of this chapter is to define the steps that are carried out when a trap is to be taken.

In effect, taking a trap can be viewed as executing an operation that branches to the required handler, with a number of side effects. The side effects are defined by the statements below. An external interrupt is treated as an EXTERN_INT exception, with only debug interrupts being handled differently.

At the trap point, the ST231 transfers execution to the trap handler, starting at the address held in the HANDLER_PC control register, and saves the execution state as detailed in *Section 5.3: Saved execution state*. All operations issued before the trapping bundle are allowed to complete. All operations issued after and including the trapping bundle are discarded. The architectural state, with the exception of saved execution state, is exactly that at the trap point. Hence ST231 interrupts and exceptions can be considered precise.

Traps are handled strictly (in order), and indivisibly with respect to the bundle stream.

5.2 Exception handling

Due to the fact that more than one operation can execute at the same time, it is possible to have more than one exception thrown in a bundle. In this case, only the highest priority exception is passed to the handler.

5.3 Saved execution state

Directly following a trap the saved execution state defines the reason for the trap and the precise trap point in the execution flow of the processor. Control registers store these values for use by the handler routine.

Taking an exception can be summarized as:

```
NEXT_PC ← HANDLER_PC;    // Branch to the exception handler

EXCAUSE ← HighestPriority();        // Store information
EXADDRESS ← ExceptAddress(EXCAUSE);// for the handler

SAVED_PSW ← PSW;         // Save the PSW and PC
SAVED_PC ← BUNDLE_PC;

PSW[USER_MODE] ← 0;      // Enter supervisor mode
PSW[INT_ENABLE] ← 0;     // Disable interrupts
PSW[IBREAK_ENABLE] ← 0; // Disable instruction breakpoints
PSW[DBREAK_ENABLE] ← 0;   // Disable data breakpoints
```

Where the function `HighestPriority` returns the highest priority exception from those that have been thrown, refer to *Section 5.6*. The `ExceptAddress` function defines the value that is stored into the EXADDRESS control register. Its return value is either 0 or the address of the data or instruction which has triggered the exception.

Therefore:

```
variable ← ExceptAddress(exception);
```

is equivalent to:

```
IF ((exception = DBREAK) OR
(exception = MISALIGNED_TRAP) OR
    (exception = CREG_NO_MAPPING) OR
    (exception = CREG_ACCESS_VIOLATION) OR
    (exception = DTLB) OR
    (exception = ITLB)) THEN
  variable ← value;
ELSE
  variable ← 0;
```

Where `value` is the optional argument that is passed to THROW (see *Section 17.3.5: Exceptions on page 127*) when the exception was generated.

The core uses a **rfi** (return from interrupt) operation to recommence execution at the trap point. An **rfi** operation causes the following state updates:

```
PC ← SAVED_PC;                  // Address execution control is
                                // transferred to by rfi. Can be
                                // altered during the exception
                                // handler routine.

PSW ← SAVED_PSW;                // Restore saved_psw. Can be
                                // altered during the exception
                                // handler routine.

SAVED_PC ← SAVED_SAVED_PC;      // Restore previous saved_pc

SAVED_PSW ← SAVED_SAVED_PSW;    // Restore previous saved_psw
```

5.4 Interrupts

All interrupts are effectively treated by the ST231 as an exception of type EXTERN_INT. Individual interrupt lines are indicated by registers in the interrupt controller. See *Chapter 12: Interrupt controller on page 80*.

5.5 Debug interrupt handling

Refer to *Chapter 13: Debugging support (TAPLink) on page 87*.

5.6 Exception types and priorities

The EXCAUSENO control register gives the cause of the last exception. Since only one exception is thrown at a time, simultaneous exceptions are prioritized. The bit fields for this register are listed in *Table 2.*

Table 2. EXCAUSENO bit fields

Name	Bit(s)	Writable	Reset	Comment
EXCAUSENO	[4:0]	RW	0x0	Specifies the exception number.
Reserved	[31:5]	RO	0x0	Reserved.

For backward compatibility, the exception cause is also available as a bit-field by reading the EXCAUSE register. The EXCAUSE register is read only and always returns `1 << EXCAUSENO_EXCAUSENO`.

Table 3 shows the possible exceptions and the value in the EXCAUSENO bitfield of the EXCAUSENO control register that each corresponds to. The table is listed in exception priority order starting with the highest priority.

Table 3. EXCAUSENO_EXCAUSENO values

Name	Value	Comment
STBUS_IC_ERROR	0	The instruction cache caused a bus error.
STBUS_DC_ERROR	1	The data cache caused a bus error.
EXTERN_INT	2	There was an external interrupt.
IBREAK	3	An instruction address breakpoint has occurred.
ITLB	4	An instruction related TLB exception has occurred.
SBREAK	5	A software breakpoint was found.
ILL_INST	6	The bundle could not be decoded into legal sequence of operations or a privileged operation is being issued in user mode.
SYSCALL	7	System call.
DBREAK	8	A breakpoint on a data address has been triggered.
MISALIGNED_TRAP	9	The address is misaligned and misaligned accesses are not supported.
CREG_NO_MAPPING	10	The load or store address was in control register space, but no control register exists at that exact address.
CREG_ACCESS_VIOLATION	11	A store to a control register was attempted whilst in user mode.
DTLB	12	A data related TLB exception has occurred.
RESERVED	13	Reserved
SDI_TIMEOUT	14	One of the SDI interfaces timed out while being accessed.

5.6.1 Illegal instruction definition

An illegal instruction exception is caused when an illegal bundle is executed. A legal bundle and all syllables contained in it must conform to the restrictions as detailed in *Chapter 18: Instruction set on page 137*.

In particular, a legal bundle and all the syllables it contains must conform to the following.

- All syllables must be valid operations or an immediate extension.
- A bundle must have a stop bit that is, four zero stop bits are illegal.
- Unused opcode fields must be set to zero, including bit 30.
- Any **branch**, **call**, **rfi**, **pswset** and **pswclr** operation must appear as the first syllable of a bundle.
- Multiply operations must appear at odd word addresses.
- Immediate extensions must appear at even word addresses.
- Immediate extensions must associate with an operation that is in the same bundle and has an immediate format that can be extended.
- There may be no more than one immediate extension associated with a single operation.
- A privileged operation can only be executed in supervisor mode. This includes **rfi**, **pswset**, **pswclr**, **prginspg** and **prgins**.
- There can only be one operation requiring the load/store unit in each bundle. This includes **sync**, **prgset**, **prgadd**, **prginspg**, **pswset**, **pswclr**, **rfi**, **ldb**, **ldh**, **ldw**, **stb**, **sth** and **stw**.
- The **sbrk** operation must have the stop bit set.
- Destination registers in a bundle have to be unique, with the exception of R0.
- **ldb**, **ldh** and **mul** operations must not have R63 as a destination register.
- **prgins** and **syscall** must be alone in a bundle.

5.7 Speculative load considerations

Speculative (or dismissible) loads execute as normal loads except in the following cases.

- The address is in a region where a speculative **load** may be destructive. In this case, the SCU (see *Section 6.6: Speculative control unit (SCU) on page 46*) should be set up to prevent speculation to this region. In this case, a zero is always returned and no access is made to the memory.
- A normal **load** would cause an exception. Generally, in this case, the **load** is considered to have been incorrectly speculated and the data is not utilized in the correct execution of the program. Zero is returned by default.
- If a dismissible **load** causes a bus error then a bus error exception is always thrown. The TLB and/or SCU should always be set up to prevent dismissible loads from causing bus errors. See *Chapter 6: Memory translation and protection on page 31*.

5.7.1 Misaligned implementation

Application or system software may require misalignment support, with misaligned accesses being correctly interpreted by the exception handler. To improve speculative load support for misaligned addresses, a control value PSW[SPECLOAD_MALIGNTRAP_EN] can be set which causes speculative loads to trap on misaligned addresses rather than returning zero, see *Section 3.4: Program status word (PSW) on page 19*.

6 Memory translation and protection

The ST231 provides full memory translation and protection by means of a translation lookaside buffer (TLB).

The TLB enables the ST231 to run memory-managing operating systems (such as Linux). It also provides a level of backward compatibility for the instruction protection unit (IPU) and data protection unit (DPU) functions when running a non-memory managed OS (for example, OS21).

The ST231 memory management system allows multiple virtual address spaces. Each virtual address space has associated with it an address space identifier (ASID).

The ST231 memory management system allows memory pages to be marked with three different policies: cached, uncached and write combining uncached, as defined in *Table 9*.

6.1 TLB overview

The ST231 has a small instruction TLB (ITLB), a small data TLB (DTLB) and a larger unified TLB (UTLB).

The ITLB performs instruction address translations and acts as a cache for address translations stored in the UTLB. When the ITLB misses it automatically updates from the UTLB.

The DTLB performs data address translations and acts as a cache for address translations stored in the UTLB. When the DTLB misses it automatically updates from the UTLB.

When the UTLB is changed, the ITLB and DTLB are not updated. The ITLB and DTLB can be flushed under software control by means of the TLB_CONTROL register, see *Section 6.4.8: TLB_CONTROL on page 40*.

The ITLB and DTLB act as small caches that keep copies of the currently active translations. Only translations that are shared or match the current ASID are loaded into the ITLB and DTLB.

Table 4 provides details of the TLB configuration of the ST231.

Table 4. TLB information

Item	Size	Comment
DTLB	8 entries	Fully associative buffer with LRU replacement.
ITLB	4 entries	Fully associative buffer with LRU replacement.
UTLB	64 entries	Fully associative buffer which is managed by the software.

The UTLB size can be determined either by reading the core version register (using a lookup table) or reading the TLB_REPLACE register after reset.

6.2 Address space

This section deals with physical and virtual addresses.

6.2.1 Physical addresses

The ST231 TLB supports 32-bit addresses providing up to 4 Gbyte of physical address space. The layout of the TLB registers allows future variants to support up to 45 bits of physical address space.

6.2.2 Virtual addresses

Virtual addresses are 32-bits. The TLB performs the mapping from virtual to physical addresses using one of the following page sizes: 8 Kbyte, 4 Mbyte and 256 Mbyte.

Control registers are accessed by virtual addresses only; virtual addresses corresponding to control registers are not translated. The (virtual) addresses of the control registers are valid physical addresses; any access to these physical addresses will be made to the memory subsystem in the usual way. If the TLB is disabled then the untranslated address will access control registers.

6.3 Caches

The caches are virtually indexed and physically tagged. The cache tag RAM lookup occurs in parallel with the TLB lookup.

6.3.1 Instruction cache organization

Instruction cache addressing is illustrated in *Figure 2*.

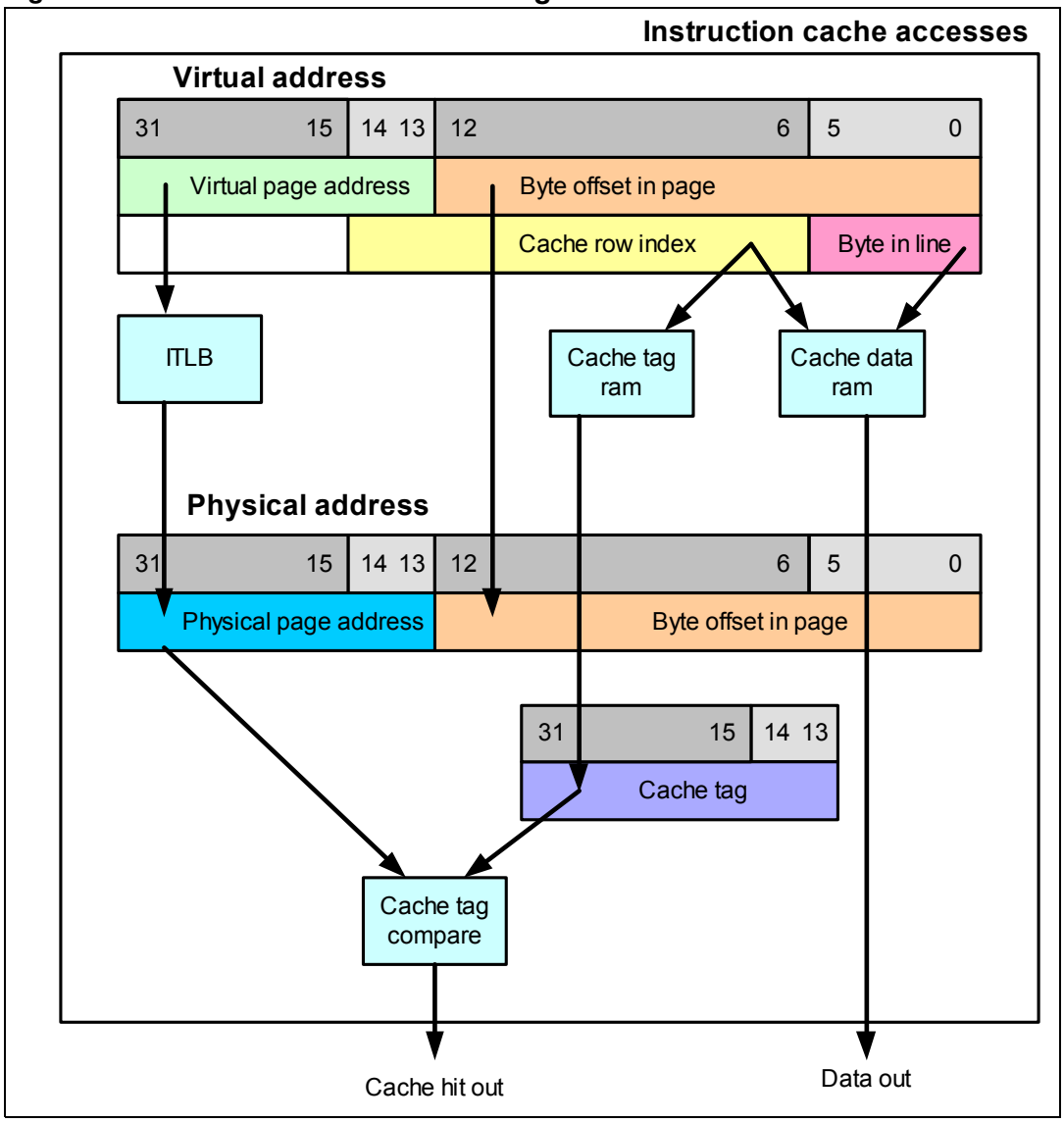
The instruction cache is 32 Kbytes direct mapped and built from 512 x 64 byte lines.

The virtual address bits [14:06] are used to index the instruction cache RAMs.

Virtual address bits [31:13] are sent to the ITLB for translation. The translated physical address bits [31:13] from the ITLB is then compared against the instruction cache tag.

Virtual address bits [05:00] are used to select the correct bytes from the cache line.

Figure 2. Instruction cache addressing



6.3.2 Data cache organization

Instruction cache addressing is illustrated in *Figure 3*.

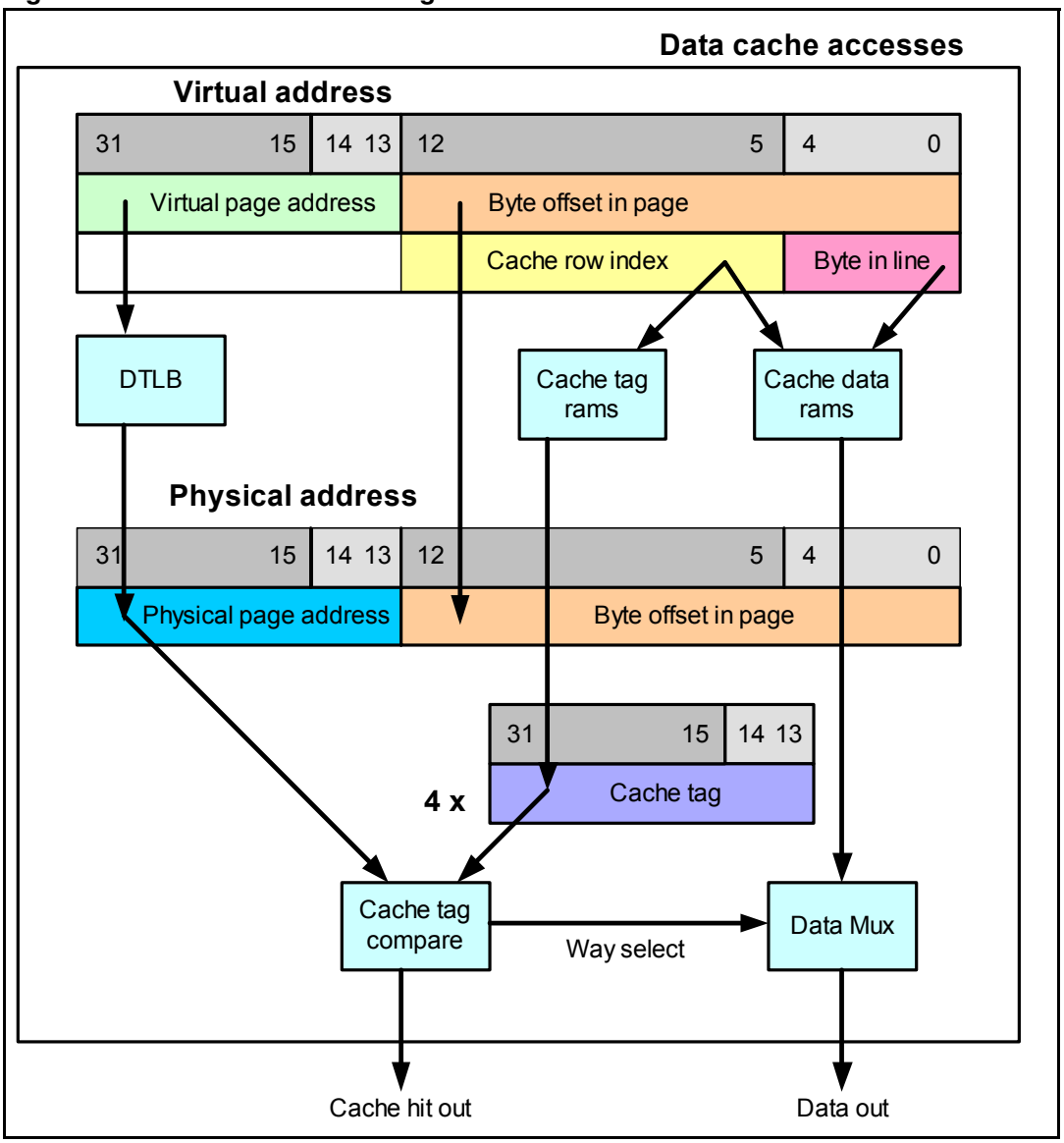
The data cache is 32 Kbytes four way set associate and built from 4 x 256 x 32 byte lines.

The virtual address bits [12:05] are used to index the data cache RAMs.

Virtual address bits [31:13] are sent to the DTLB for translation. The translated physical address bits [31:13] from the DTLB are then compared against the data cache tag.

Virtual address bits [04:00] are used to select the correct bytes from the cache line.

Figure 3. Data cache addressing



6.4 Control registers

A full list of control registers is provided in *Chapter 9: Control registers on page 67*.

6.4.1 PSW

The TLB can be enabled and disabled by a bit in the PSW, see *Chapter 3: Architectural state on page 19*.

While address translation is disabled (TLB_ENABLE = 0):

- virtual addresses are not translated and are used directly as the physical address
- all data accesses are made uncached
- no TLB exceptions are thrown

6.4.2 TLB_INDEX

Table 5 shows the mapping for the TLB_INDEX register.

Table 5. TLB_INDEX bit fields

Name	Bit(s)	Writable	Reset	Comment
ENTRY	[7:0]	RW	0x0	Determines which of the 64 TLB entries is mapped to the TLB_ENTRYx registers. Writing a value to this register that is greater than the maximum UTLB entry available has no effect (the entry is not updated).
RESERVED	[31:8]	RO	0x0	Reserved.

When the TLB_INDEX register is written, subsequent read/writes to the TLB_ENTRYX registers are to the indicated UTLB entry.

6.4.3 TLB_ENTRY0

This register maps bits [31:00] of the TLB entry. The entry is chosen by writing to the TLB_INDEX register. *Table 6* lists the fields of the TLB_ENTRY0 register; the fields are described in subsequent tables.

Table 6. TLB_ENTRY0 bit fields

Name	Bit(s)	Writable	Reset	Comment
ASID	[7:0]	RW	0x0	Indicates which address space this page belongs to.
SHARED	8	RW	0x0	Page shared by multiple address spaces (ASIDs).
PROT_SUPER	[11:9]	RW	0x0	A three bit field that defines the protection of this region in supervisor mode.
PROT_USER	[14:12]	RW	0x0	A three bit field that defines the protection of this region in user mode.

Table 6. TLB_ENTRY0 bit fields (continued)

Name	Bit(s)	Writable	Reset	Comment
DIRTY	15	RW	0x0	Page is dirty. When this bit is 0 write accesses to this page (when write permission is allowed) cause a TLB_WRITE_TO_CLEAN exception. When this bit is 1 writes to this page (when write permission is allowed) are permitted.
POLICY	[19:16]	RW	0x0	Cache policy for this page.
SIZE	[22:20]	RW	0x0	Size of this page (also used to disable the page).
PARTITION	[24:23]	RW	0x0	Data cache partition indicator.
RESERVED	[31:25]	RO	0x0	Reserved.

Writing zero to this register disables the page.

Table 7 lists the possible values of the POLICY field.

Table 7. TLB_ENTRY0_POLICY values

Name	Value	Comment
UNCACHED	0	Uncached mode. Reads and write that miss the cache are uncached.
CACHED	1	Cached mode. Reads that miss the cache cause the cache to be filled. Writes that hit the cache are written into the cache. Writes that miss the cache are sent to the write buffer.
WCUNCACHED	2	Write combining uncached. Writes that miss the cache are sent to the write buffer. Reads that miss the cache are uncached.
Reserved	3	Reserved (on the ST230 reserved cache policies default to uncached).
Reserved	4	Reserved (on the ST230 reserved cache policies default to uncached).
Reserved	5	Reserved (on the ST230 reserved cache policies default to uncached).
Reserved	6	Reserved (on the ST230 reserved cache policies default to uncached).
Reserved	7	Reserved (on the ST230 reserved cache policies default to uncached).

Table 8 lists of the possible values of the SIZE field.

Table 8. TLB_ENTRY0_SIZE values

Name	Value	Comment
DISABLED	0	Page is disabled.
8K	1	8 KByte page.
4MB	2	4 MByte page.
256MB	3	256 MByte page.
Reserved	4	Reserved (on the ST230 reserved page sizes disable the page).
Reserved	5	Reserved (on the ST230 reserved page sizes disable the page).
Reserved	6	Reserved (on the ST230 reserved page sizes disable the page).
Reserved	7	Reserved (on the ST230 reserved page sizes disable the page).

Table 9 lists of the possible values of the PARTITION field:

Table 9. TLB_ENTRY0_PARTITION values

Name	Value	Comment
REPLACE	0	Place in the way specified by the replacement counter and increment the counter.
WAY1	1	Place in the way 1 only.
WAY2	2	Place in the way 2 only.
WAY3	3	Place in the way 3 only.

Table 10 lists of the possible values of the PROT_USER and PROT_SUPER fields:

Table 10. TLB_PROT values

Name	Value	Comment
EXECUTE	1	Execute permission.
READ	2	Read (prefetch and purge) permission.
WRITE	4	Write permission.

6.4.4 TLB_ENTRY1

This register allows access to bits [63:32] of the TLB entry. The entry is chosen by writing to the TLB_INDEX register. The fields in this register are listed in *Table 11.*

Table 11. TLB_ENTRY1 bit fields

Name	Bit(s)	Writable	Reset	Comment
VADDR	[18:0]	RW	0x0	The upper 19 bits of the virtual address. For 4Mbyte pages only the upper 10 bits of this field are significant. For 256 Mbyte pages only the upper 4 bits of this field are significant.
RESERVED	[31:19]	RO	0x0	Reserved.

6.4.5 TLB_ENTRY2

This register allows access to bits [95:64] of the TLB entry. The entry is chosen by writing to the TLB_INDEX register. The fields in this register are listed in *Table 12.*

Table 12. TLB_ENTRY2 bit fields

Name	Bit(s)	Writable	Reset	Comment
PADDR	[18:0]	RW	0x0	The upper 19 bits of the physical address. For 4 Mbyte pages only the upper 10 bits of this field are significant. For 256 Mbyte pages only the upper 4 bits of this field are significant.
Reserved	[31:19]	RO	0x0	Reserved.

6.4.6 TLB_ENTRY3

This register maps bits [127:96] of the TLB entry. The entry is chosen by writing to the TLB_INDEX register. The fields in this register are listed in *Table 13.*

Table 13. TLB_ENTRY3 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	RO	0x0	Reserved.

6.4.7 TLB_REPLACE

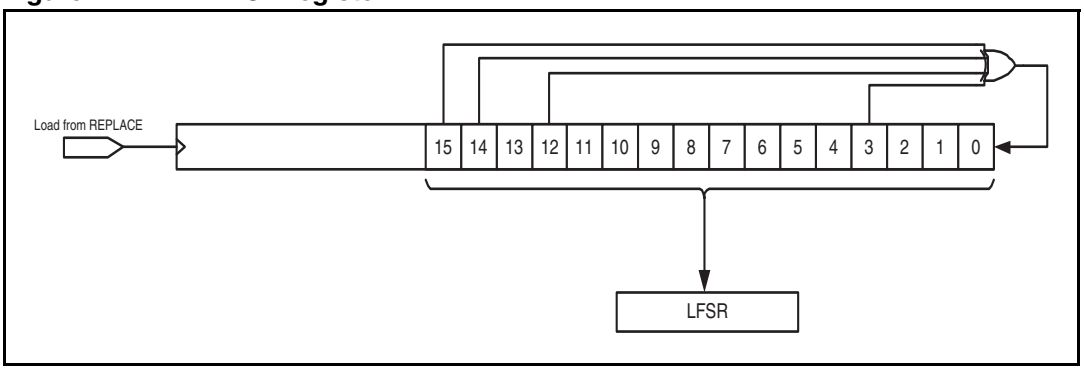
Table 14 shows the mapping of the TLB_REPLACE register.

Table 14. TLB_REPLACE bit fields

Name	Bit(s)	Writable	Reset	Comment
LFSR	[15:0]	RW	0xFFFF	Random number used to determine which entry to replace next.
LIMIT	[23:16]	RW	0x40	Number of TLB entries that can be replaced.
Reserved	[31:24]	RO	0x00	Reserved.

Figure 4 shows the structure of the REPLACE register.

Figure 4. REPLACE register



Software uses the replacement register to randomly decide which TLB entry to replace. The value of the REPLACE field is generated in a pseudo-random manner using a 16-bit linear feedback shift register (LFSR) generating a maximum length sequence (taps on bits 3, 12, 14 and 15).

A read from the TLB_REPLACE register returns the current LFSR and LIMIT values, the LFSR is then clocked to generate a new value. The current value of the LFSR field can be changed by writing to the TLB_REPLACE register.

The LIMIT field is reset to the number of entries in the TLB[a]. The LIMIT field can be changed by writing the TLB_REPLACE register. To reserve a number of entries for a fixed mapping, software sets the LIMIT field to less than the number of entries available to the TLB.

The LIMIT field is not used by the hardware but is included to allow the software to quickly determine the next TLB entry to replace. A suggested replacement algorithm is as follows:

```
unsigned replace, lfsr, limit, index;

// Read replace register to get LIMIT and LFSR
replace = VOLUINT(LXTLB_REPLACE);

// Extract fields
lfsr = LXTLB_REPLACE_LFSR(replace);
limit = LXTLB_REPLACE_LIMIT(replace);

// Decide which entry to replace
index = (lfsr * limit) >> 16;

// Select the correct entry
VOLUINT(LXTLB_INDEX) = index;
```

Note: The **mullhu** *instruction can be used to extract the LFSR and LIMIT fields from the* TLB_REPLACE *register and perform the multiply. The result is then shifted right by 16 bits to obtain the entry number to replace.*

a. This depends on which core is implemented.

6.4.8 TLB_CONTROL

Table 15 shows the mapping of the TLB_CONTROL register.

Table 15. TLB_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
ITLB_FLUSH	0	RW	0x0	Writing a 1 to this bit flushes the entire ITLB. Writing 0 to this bit has no effect. This bit always reads as zero.
DTLB_FLUSH	1	RW	0x0	Writing a 1 to this bit flushes the entire DTLB. Writing 0 to this bit has no effect. This bit always reads as zero.
Reserved	[31:2]	RO	0x0	Reserved.

Before the ITLB or DTLB are flushed, the hardware ensures that all outstanding writes to the UTLB have completed.

6.4.9 TLB_ASID

Table 16 shows the mapping of the TLB_ASID register.

Table 16. TLB_ASID bit fields

Name	Bit(s)	Writable	Reset	Comment
ASID	[7:0]	RW	0x0	Address space identifier. Writes to this register also cause the ITLB and DTLB to be flushed.
Reserved	[31:8]	RO	0x0	Reserved.

Coherency

After making the following changes the software must ensure coherency by:

- changing the current ASID
- updating the UTLB

Changing the ASID

Changing the current ASID requires that all instructions that may be affected by the change are flushed from the execution pipeline and the instruction buffer. This should be achieved by:

```
syncins;;
```

If it can be guaranteed that no instructions are affected by the change in ASID value for 8 bundles after the change then this may be omitted.

Updating the UTLB

Updating the UTLB requires 6 cycles for the change to take effect. Updating the UTLB does not automatically update the ITLB or DTLB.

If the meaning of a virtual address is changed (including mapping to a different physical address or changing other properties), instructions in the pipeline or instruction buffer using old translations could be incoherent. In this case the ITLB or DTLB must be flushed to ensure coherency. The execution pipeline and instruction buffer must also be flushed.

Clearing the instruction buffer and the execution pipeline can be achieved with **syncins**.

If updating the UTLB in an exception handler, returning from the handler with an **rfi** clears the instruction buffer so **syncins** is not also necessary.

The recommended sequences for ensuring coherency are shown in *Table 17*. **$r1** contains the value 1-3 depending on which of the ITLB and/or DTLB require flushing, see *Section 6.4.8: TLB_CONTROL on page 40*.

Table 17. Ensuring coherency after UTLB updates

	Properties of VA changed	Properties of VA not changed

Normal	`stw TLB_CONTROL[$r0] = $r1;;`
	`nop;;`
	`syncins;;`
		`nop;;`
		`nop;;`
		`syncins;;`

Within exception handler	`stw TLB_CONTROL[$r0] = $r1;;` `rfi;;`	`nop;;` `rfi;;`

The normal sequence allowing the properties of the virtual address to change may be used to ensure coherency in any case.

6.4.10 TLB_EXCAUSE

When the ST231 raises a TLB exception, the TLB_EXCAUSE register is updated. The possible exceptions are listed in *Table 18*.

Table 18. TLB_EXCAUSE_CAUSE values

Name	Value	Comment
NO_MAPPING	0	No mapping was found. (The UTLB had no mapping for the virtual address. The given page is not in the UTLB and the DTLB (for data accesses) or the ITLB (for instruction accesses)
PROT_VIOLATION	1	An attempt has been made to violate the permissions of a page. The given page may be in the ITLB, DTLB, UTLB or any combination of the above.
WRITE_TO_CLEAN	2	A write to a clean page has occurred. This allows the software managing the UTLB to update the master copy of the table kept in memory and to handle any shared pages.
MULTI_MAPPING	3	There were multiple hits in the UTLB. The software managing the TLB should ensure that this does not happen.

When a PROT_VIOLATION or WRITE_TO_CLEAN exception is thrown for data accesses the given page may be in the ITLB, DTLB, UTLB or any combination of the above.

It is possible for a page to be held in the ITLB or DTLB but not in the UTLB if the software chose not to purge the ITLB/DTLB when a page was replaced.

When a MULTI_MAPPING exception is thrown, the virtual address maps to more than one page in the UTLB. In this case the IN_UTLB and INDEX fields of the TLB_EXCAUSE and

the EXADDRESS registers contain the virtual address of the syllable or data which triggered the exception.

When a TLB exception is taken, the software can determine if the given page is in the UTLB by checking the IN_UTLB bit.

Table 19 describes the bit fields within the TLB_EXCAUSE register.

Table 19. TLB_EXCAUSE bit fields

Name	Bit(s)	Writable	Reset	Comment
INDEX	[7:0]	RW	0x0	TLB index of excepting page.
Reserved	[15:8]	RO	0x0	Reserved
CAUSE	[17:16]	RW	0x0	Cause of current TLB exception.
SPEC	18	RW	0x0	When 1 indicates that this exception was caused by either a purge address or speculative load instruction.
WRITE	19	RW	0x0	When 1 indicates that this exception was caused by an attempted write to a page (store). When 0 indicates that this exception was caused by an attempted read or purge of a page.
IN_UTLB	20	RW	0x0	When 1 the exception address is in the UTLB and the INDEX field is valid. When 0 the exception address was not in the UTLB and the INDEX field is invalid.
Reserved	[32:21]	RO	0x0	Reserved.

6.5 TLB description

The TLB functionality is controlled completely by accessing the provided control registers.

6.5.1 Reset

After reset, the contents of the UTLB are undefined. Before the TLB is enabled all entries must be programmed (or cleared) to prevent undefined behavior.

6.5.2 UTLB arbitration

The DTLB, ITLB and TLB control registers have to arbitrate for access to the UTLB. The priority of UTLB accesses are as follows:

1 (highest) TLB control register access.

2 DTLB.

3 (lowest) ITLB.

6.5.3 Exceptions

When the TLB throws an exception it jumps to the exception vector and updates the TLB_EXCAUSE, EXADDRESS and EXCAUSENO registers.

When a DTLB exception is thrown, the EXADDRESS register contains the virtual effective address that caused the exception.

When an ITLB exception is thrown, the EXADDRESS register contains the virtual address of the syllable that caused the exception. In the case of possible multiple ITLB exceptions, the exception with the lowest syllable address is thrown.

The SAVED_PC and SAVED_PSW stack are also updated in the same way as other traps.

As noted above, the TLB_EXCAUSE register indicates the nature of the ITLB or DTLB exception.

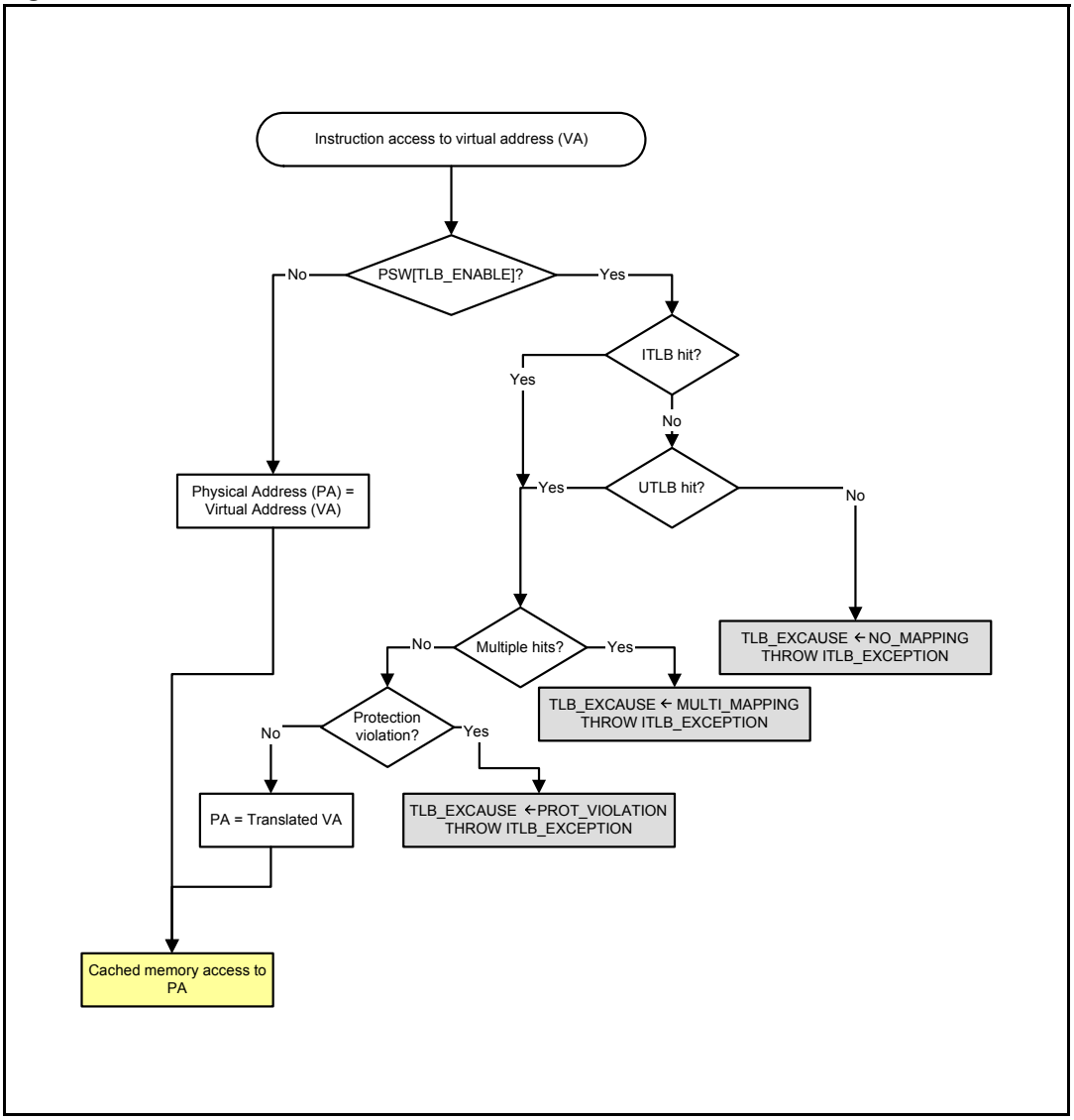
Note: Misaligned loads and control register violations are not considered TLB exceptions.

Details of the EXCAUSENO register can be found in *Section 5.6: Exception types and priorities on page 28*.

6.5.4 Instruction accesses

Instruction accesses are always cached (the cache policy is ignored). The procedures for accessing instructions are summarized in *Figure 5*.

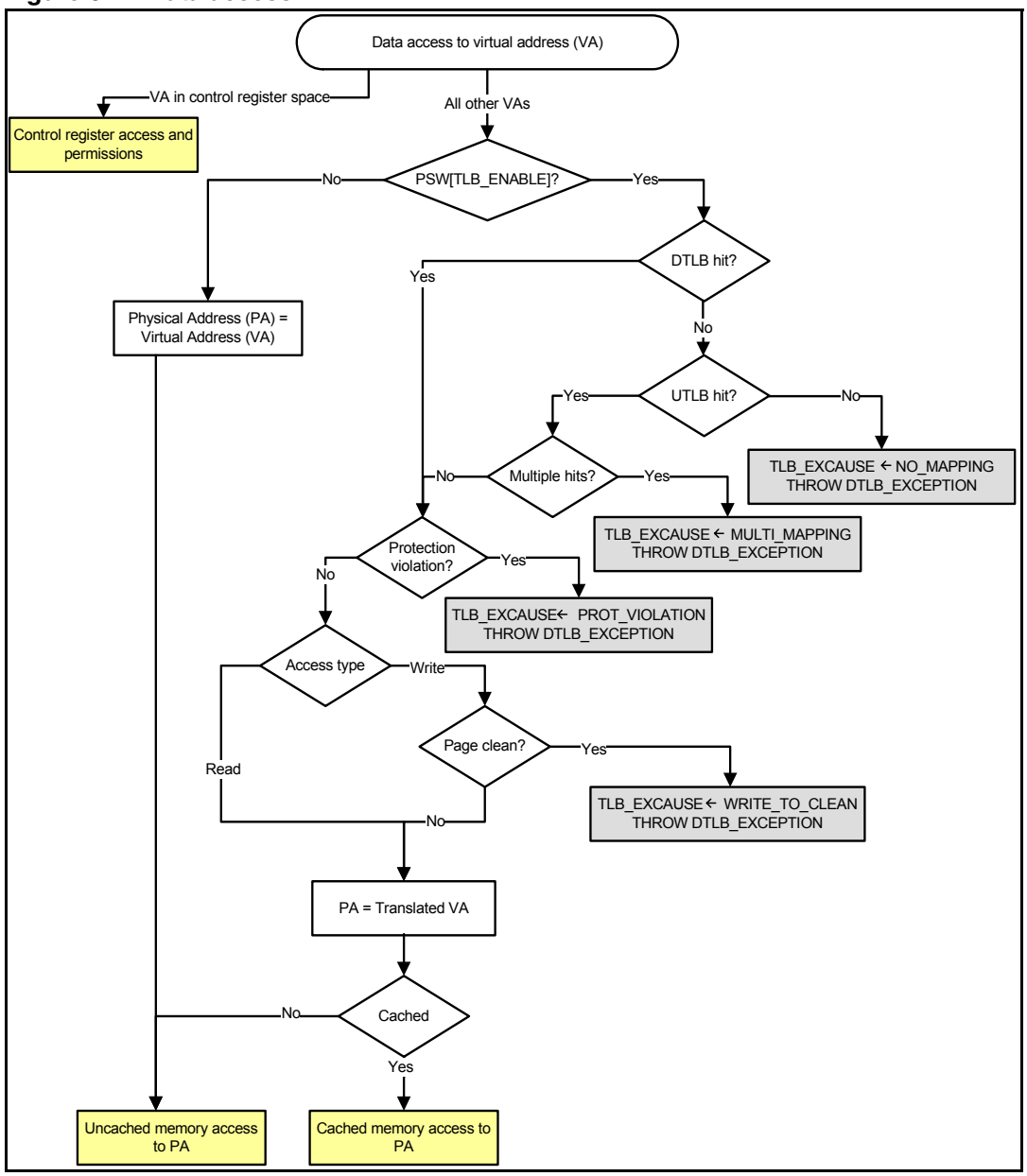
Figure 5. Instruction access



6.5.5 Data accesses

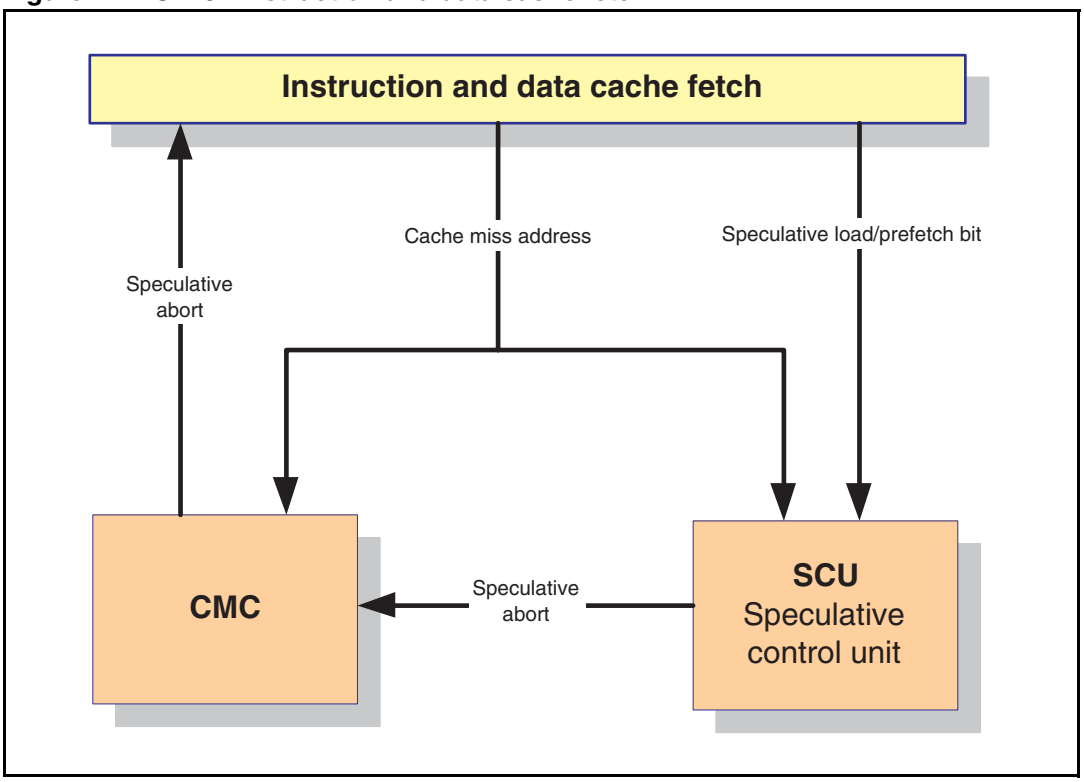
The procedures for reading and writing data are summarized in *Figure 6*.

Figure 6. Data access



6.6 Speculative control unit (SCU)

Figure 7. ST231 instruction and data cache fetch



The SCU filters physical speculative **load** addresses (both cached and uncached) and prefetches that miss the cache to ensure that speculative bus requests are not sent out to peripherals and unmapped memory regions.

The SCU supports four regions of memory aligned to the smallest TLB page size (8 Kbyte). If the physical address of the speculative **load/prefetch** address falls within one of the four supported regions, the bus access is allowed, otherwise the SCU aborts the speculative load/prefetch and zero is returned or the prefetch is cancelled.

The regions are configured using the SCU_BASEx and SCU_LIMITx control registers. A region may be disabled by setting the base to be larger than the limit.

The SCU resets so each of the four regions cover the whole of memory. This allows speculative loads to be issued before the SCU has been initialized.

6.6.1 SCU_BASEx

The SCU base register defines the physical start address of the region where speculative loads/prefetches are permitted. This region is aligned to the smallest page size (by virtue of the read only zero bits). The base address is inclusive, so setting BASE == LIMIT defines an 8 Kbyte region.The fields of the SCU base register are listed in *Table 20*.

Table 20. SCU_BASE0 bit fields

Name	Bit(s)	Writable	Reset	Comment
BASE	[18:0]	RW	0x0	Upper 19 bits of the base of this region.
RESERVED	[31:19]	RO	0x0	Reserved.

6.6.2 SCU_LIMITx

The SCU limit register defines the physical end address of the region where speculative loads are permitted. This region is aligned to the smallest page size (by virtue of the read only zero bits). The limit address is inclusive, so setting BASE == LIMIT defines an 8 Kbyte region. The fields of the SCU limit register are listed in *Table 21*.

Table 21. SCU_LIMIT0 bit fields

Name	Bit(s)	Writable	Reset	Comment
LIMIT	[18:0]	RW	0x7FFFF	Upper 19 bits of the limit of this region.
RESERVED	[31:19]	RO	0x0	Reserved.

6.6.3 Updates to SCU registers

Any changes to SCU registers take effect for future bus transactions. To ensure that all STBus transactions prior to an SCU change are made with the old settings, and all STBus transactions subsequent to an SCU change are made with the new settings, the program must execute a **sync** instruction before updating the SCU registers.

7 Memory subsystem

This chapter describes the operation of the ST231 processor memory subsystem. The memory subsystem includes the following components:

- caches
- protection units
- write buffer
- prefetch cache
- translation lookaside buffer (TLB)
- the core memory controller (CMC)

The memory subsystem is split broadly into two parts: the instruction side (I-side) and the data side (D-side). The CMC interfaces these two parts to the STBus port. The I-side, containing the instruction cache, supports the fetching of instructions. The D-side, containing the data cache, prefetch cache and write buffer, support the storing and loading of data.

The TLB performs memory address translation and protection. The function of the TLB is detailed in *Chapter 6: Memory translation and protection on page 31*.

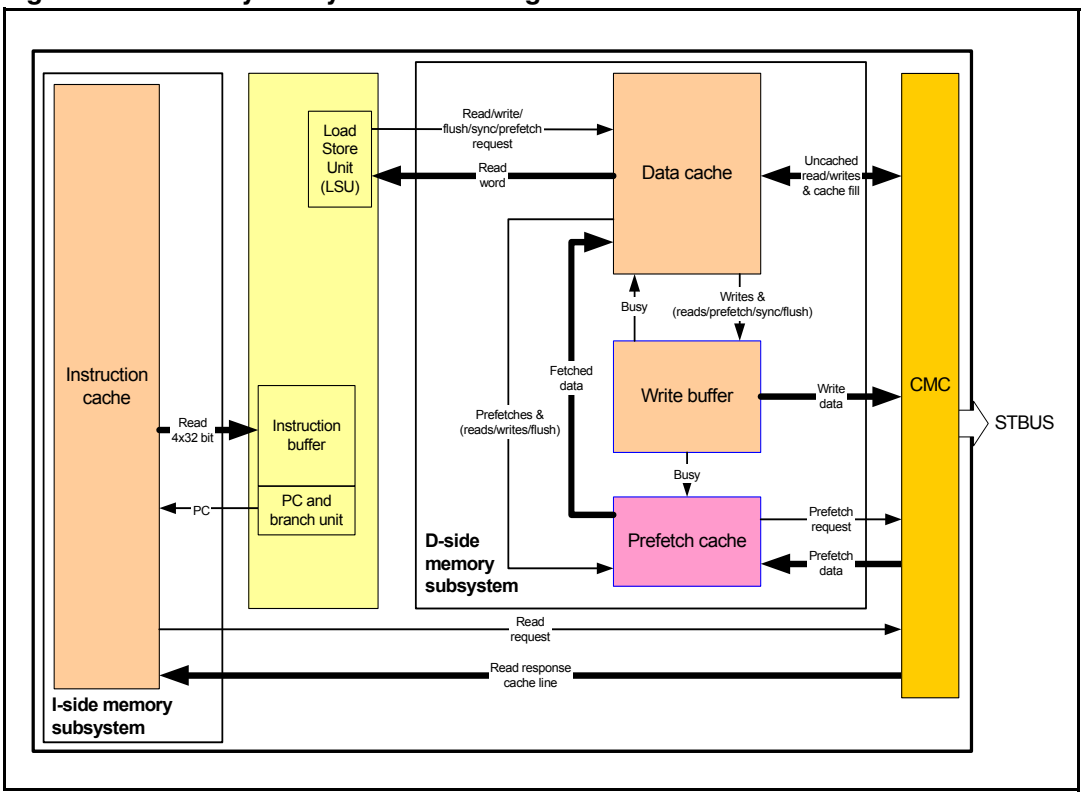
The ST231 ensures that data accesses are coherent with other data accesses. There is no guarantee of coherency between instruction and data accesses (see *Section 7.5.2: Coherency between I-side and D-side on page 59*) or between the core and external memory. To ensure coherency data must be purged from the core as described later in this chapter.

The function of the streaming data interface (SDI) is described in *Chapter 8: Streaming data interface on page 61*.

7.1 Memory subsystem

This section describes the memory subsystem shown in *Figure 8*.

Figure 8. Memory subsystem block diagram



7.2 I-side memory subsystem

Within the ST231, the instruction buffer is responsible for issuing instructions to the processor core. The instruction cache uses the CMC to fetch cache lines from memory, and sends groups of up to four operations to the instruction buffer.

7.2.1 Instruction buffer

The instruction buffer attempts to fetch ahead in the instruction stream in order to keep its buffer full. When a branch is taken the instruction buffer is invalidated and a fetch started from the target address.

After a branch, the instruction buffer takes one cycle to fetch the next bundle from the cache, this means the ST231 stalls for one cycle. If the branch is to a bundle that spans two cache lines, then it takes two cycles to fetch the bundle and thus the ST231 stalls for two cycles.

7.2.2 Instruction cache

Instructions are always cached; there is no support for uncached instruction fetching. Self modifying code (loaders for example) must invalidate the cache explicitly.

The instruction cache is a 32 Kbyte direct mapped cache. The cache is made up from 512 sets. Each set contains one 64-byte line.

When a virtual (byte) address is submitted to the cache the address bits are used as follows:

- Bits [05:02] select the word offset within the line
- Bits [14:06] select the set that could contain the required cache line
- Bits [31:13] are translated by the TLB. The resulting physical tag is used to check if the required line is in the cache

The instruction cache receives fetch requests from the instruction buffer and returns a group of up to four 32-bit operations (16 bytes).

When syllables are requested from the cache it uses the address to determine whether they are already present in the cache. If the syllables are not in the cache, they are fetched from memory and stored into the cache, during which time the processor stalls. The requested instruction bundle is then returned to the instruction buffer.

Invalidating the entire instruction cache

To invalidate the instruction cache safely, the core must execute the following two operations.

- The first is **prgins**, which invalidates the whole instruction cache and causes any subsequent instruction fetches to be made from memory rather than from the cache.
- The second is **syncins**, which ensures that the next bundle is fetched with an invalidated cache, and therefore from memory; if this operation is not performed the subsequent bundle might have been prefetched into the instruction buffer and might not correspond to the instruction in memory. The **syncins** operation is a pseudo operation implemented as a **pswset**.

*Note: Both **prgins** and **syncins** are privileged operations (which means that they can only be executed in supervisor mode).*

Invalidating the instruction cache by page

The instruction cache can also be invalidated in 8 Kbyte pages by means of the **prginspg** instruction.

The **prginspg** operation takes a base and an immediate offset. The base is added to the offset to produce the effective address (as for other load/store instructions). The effective address is then used by the instruction cache to perform the purge.

The effective address is split into two parts. Bits [31:13] specify the upper bits of the 8 Kbyte physical address of the page to be purged. In addition, bits [12:0] are used to specify any ambiguous address bits of the virtual cache address that may contain entries for the physical page. This ambiguity occurs when the instruction cache set(s) are larger than the purgeable page size.

The single way cache is 32 Kbyte and the purgable page size only 8 Kbyte. As a result, two bits of virtual address need to be specified to indicate the quadrant of the indexed cache in which physical page entries are scanned for. The bottom two bits of the effective address (bits [14:13] of the virtual address) are used to specify the quadrant.

We recommend that the software system should provide all 13 bits of a potentially ambiguous virtual address to ensure future compatibility.

To purge a physical page which has more than one virtual mapping, multiple purge pages need to be executed (except in the case where all the ambiguous address bits are the same).

Note: ***prginspg*** *is a privileged operation (which means that it can only be executed in supervisor mode).*

7.2.3 I-side bus error

If the I-side memory subsystem causes a bus error, an STBUS_IC_ERROR exception is thrown. I-side bus errors are synchronous events which are thrown when trying to execute the bundle which causes the bus error.

A bus error invalidates the cache line.

7.3 D-side memory subsystem

All data accesses take place through the D-side memory subsystem which contains the data cache, the prefetch cache and the write buffer.

The data cache is 32 Kbyte 4-way associative with a 32-byte line. The cache is therefore made up of 256 sets. Each set contains 4 cache lines, one per way. It is operated with a fixed write-back, no allocate on write-miss policy.

When a virtual (byte) address is submitted to the cache its bits are used as follows.

- Bits [4:0] select the byte offset within the cache line.
- Bits [12:5] select the set (0-255) that could contain the required cache line.
- Bits [31:13] are translated by the TLB. The resultant physical address form the tag that is used to check if the required line is in the cache.

At most one of the write buffer, the data cache or the prefetch cache can contain a copy of the data for a particular address.

7.3.1 Load/store unit

The load/store unit (LSU) performs all data access operations. The cacheability is dependent on the address of the access and is determined by the TLB**,** see *Chapter 6: Memory translation and protection on page 31*. In addition to **load** and **store** there are operations which prefetch data, flush and synchronize the D-side memory subsystem.

The data cache sends write misses and dirty data to the write buffer, see *Write buffer on page 58*.

The write buffer combines write transactions and sends them out to memory.

7.3.2 Data cache partitioning

Data cache partitioning allows, in addition to the normal mode, the data cache to be split into a normal and either 1, 2 or 3 locked partitions, as three separate modes.

These modes allow the cache to be partitioned as:

- 32 Kbyte 4-way cache
- 24 Kbyte 3-way cache plus 8 Kbyte data RAM*
- 16 Kbyte 2-way cache plus 16 Kbyte data RAM*
- 8 Kbyte direct mapped cache plus 24 Kbyte data RAM*

Where * indicates special cases of the following:

- 24 Kbyte 3-way cache plus separate 8 Kbyte direct mapped cache
- 16 Kbyte 2-way cache plus 2 separate 8 Kbyte direct mapped caches
- 4 separate 8 Kbyte direct mapped caches

Control of partitioning is performed via two mechanisms. Firstly the machine state register (see *Section 9.4: Data cache replacement state register on page 71*) defines the number of data cache ways which are reserved (locked). Secondly a bit field in each TLB entry (see *Chapter 6: Memory translation and protection on page 31*) is used to indicate which of the sets misses are placed in (either the locked section or the rest of the cache).

7.3.3 Speculative loads

Speculative loads are defined as returning the same data as normal loads except that when a normal **load** would cause an exception, speculative **load** returns 0.

Speculative loads are handled in the following way:

- speculative loads that would cause a PROT_VIOLATION exception return 0
- if a speculative **load** misses the DTLB and maps to more than one entry in the UTLB, it causes a MULTI_MAPPING exception
- speculative loads that miss the UTLB cause a TLB_NO_MAPPING exception, if the DYNAMIC bit is set in the PSW (if this bit is clear then speculative loads that miss the UTLB return 0 without causing an exception)
- speculative loads that miss the cache are validated by the SCU, see *Section 6.6: Speculative control unit (SCU) on page 46*, if the speculative **load** does not fall into one of the valid regions in the SCU then it returns 0

7.3.4 Cached loads and stores

Cached loads and stores are performed through the data cache. The memory subsystem can optimize these operations for performance. For example, the memory subsystem can transfer more data than specified by the load (that is, a loading cache line), aggregate accesses (combining writes in write buffer) and, or re-order accesses (cache causes word accesses to be re-ordered).

The memory subsystem presents a consistent view of cached memory to the ST231 programmer, that is, a store followed by a load to the same address always returns the stored data. To guarantee ordering of accesses to external memory in cached regions, **purge** and **sync** operations must be used.

7.3.5 Uncached load and stores

Uncached loads and stores are issued to the STBus in program order. Data accessed through an uncached TLB entry or when the TLB is disabled is never brought into the data cache or the prefetch cache. See *Section 7.5.4: Cached data in uncached region on page 59*.

The precise amount of data specified in the access is transferred and the access is not aggregated with any other. The implementation does not optimize these accesses.

To guarantee that an uncached store has reached its STBus target, either a **sync** or an uncached **load** to the same bus target must be issued.

An STbus request arising from a cached (or a write-combining uncached) operation, and an STbus request arising from an uncached operation, are not guaranteed to be issued to the STbus in program order unless the operations target the same physical address. To guarantee ordering, a **sync** instruction should be inserted between the operations.

7.3.6 Prefetching data

The prefetch cache prefetches and stores data from external memory and sends it to the data cache when (and if) it is required.

A **pft** operation is a hint to the memory subsystem that the given item of data may be accessed in the future. The operation specifies a virtual address which can be prefetched by the prefetch cache. A **pft** operation may be ignored.

Prefetches are ignored (treated as **nops**) in the following cases:

- a prefetch that hits the cache
- a prefetch that does not map to a valid page in the TLB
- a prefetch to an uncached (or write combining uncached) page
- a prefetch to control register space
- a prefetch to a region that does not have read permission
- a prefetch that misses the cache but does not fall into one of the valid regions in the SCU, see *Section 6.6: Speculative control unit (SCU) on page 46*
- a prefetch that is issued when 8 other prefetches are outstanding (see below)

For this reason the only exception a prefetch can cause is a DTLB MULTI_MAPPING fault.

The prefetch cache contains eight entries. Each entry contains an entry valid bit, a prefetch address, a data valid bit and 32 bytes of data.

When a **pft** request is made and accepted, it enters the prefetch cache as an outstanding prefetch request, with the data valid bit clear. Older entries may be discarded if the prefetch cache is full. The prefetch cache attempts to access the memory system to fetch the line containing the prefetch address. When a fetch completes the data valid bit is set. The prefetch cache supports multiple outstanding memory requests.

Entries in the prefetch cache are tested when a data cache read miss occurs. If an entry match occurs and the data valid bit is set, the prefetched line is loaded into the data cache as if it were fetched from external memory. If the data valid bit is clear, the data cache stalls until the data is returned from external memory. The entry in the prefetch cache is then marked as empty and can be reused.

Entries in the prefetch cache are tested when a data cache write miss occurs. If an entry match occurs the prefetch cache entry is invalidated.

7.3.7 Purging data caches

The purge (flush and invalidate) operations are used to ensure a copy of a particular data item is not cached in the D-side of the memory subsystem.

These operations flush out the specified data. Dirty lines are written to the write buffer and the line is invalidated. Purge addresses are treated as byte aligned.

Purging data by address

The **prgadd** operation purges the specified virtual address from the data cache.

If the DYNAMIC bit in the PSW is set, and the address is not present in the UTLB, a DTLB NO_MAPPING exception is thrown.

If the DYNAMIC bit in the PSW is not set, and the address is not present in the UTLB, the purge is ignored.

If the address to be purged misses the DTLB, and it maps to more than one UTLB entry, it causes a DTLB MULTI_MAPPING exception.

If the address to be purged does not have read permission (protection fault), the **prgadd** is ignored and no exception is thrown.

Purging data by set

The **prgset** operation purges each of the four lines in the data cache set, indicated by the address operand, without checking for a cache hit. It also invalidates the entire prefetch cache.

prgset operates on a subset of the address bits. As such it can be used to purge both virtual and physical addresses.

*Note: The **prgset** operation also resets the replacement pointer in the set to way 0. This is not visible unless using data cache partitioning.*

7.3.8 D-side synchronization

This is achieved by executing the **sync** operation. Once the bundle containing the **sync** operation has completed, the following conditions hold.

- All previous **load**s, **store**s and **pft**s have completed.
- No future memory operations have started.
- The write buffer is empty, all pending writes to external memory have completed.

7.3.9 D-side bus errors

If the D-side memory subsystem causes a bus error, a STBUS_DC_ERROR exception is thrown. Bus errors are asynchronous events and are not associated with a particular operation.

In the case of writes the data has already been discarded and therefore the write is lost. The write may or may not have completed.

In the case of reads the cache line which has been allocated for the data is invalidated.

7.3.10 Operations

Table 22 lists the operations supported by the data side memory subsystem.

Table 22. Memory operations

Type	Word aligned	Half word aligned		Byte aligned	
Load	Load word	Load half unsigned	Load half signed	Load byte unsigned	Load byte signed
Load dismissible	Load word	Load half unsigned	Load half signed	Load byte unsigned	Load byte signed
Store	Store word	Store half		Store byte	
Prefetch				Prefetch	
Purge				Purge address	
				Purge set	
Sync				Sync	

It is a requirement that half word load/stores are half word aligned (2 bytes) and word load/stores are word aligned (4 bytes). Misaligned accesses cause a MISALIGNED_TRAP exception.

7.3.11 Cache policy

Table 23 details the effect of each of the above memory operations on the ST231's memory subsystem.

Note: When a cache line is purged to the write buffer it is also invalidated in the cache.

Table 23. Cache policy

Instruction	Policy	Cache	Write buffer	Prefetch	SCU	Result
Loads: **ldb, ldh, ldw**	Uncached	Miss	Miss	Miss		**Load** uncached
				Hit		Discard prefetch entry, **Load** uncached
			Hit			Flush write-buffer entry, **Load** uncached
		Hit clean				Invalidate cache line, **Load** uncached
		Hit dirty				Purge cache line to write buffer, flush write buffer, **Load** data uncached

Table 23. Cache policy (continued)

Instruction	Policy	Cache	Write buffer	Prefetch	SCU	Result
Dismissable loads **ldb.d, ldh.d, ldw.d**	Uncached	Miss	Miss	Miss	Hit	**Load** uncached
					Miss	Return 0
				Hit	Hit	Discard prefetch entry, **Load** uncached
					Miss	Discard prefetch entry, return 0
			Hit		Hit	Flush write-buffer entry, **Load** uncached
					Miss	Flush write-buffer entry, return 0
		Hit clean			Hit	Invalidate cache line, **Load** uncached
					Miss	Invalidate cache line, return 0
		Hit dirty			Hit	Purge cache line to write buffer, flush write buffer, **Load** data uncached
					Miss	Purge cache line to write buffer, flush write buffer, return 0
Stores: **stb, sth, stw**	Uncached	Miss	Miss	Miss		**Store** uncached
				Hit		Discard prefetch entry, **Store** uncached.
			Hit			Flush write buffer entry, **Store** uncached.
		Hit clean				Invalidate cache line, **Store** data uncached
		Hit dirty				Purge cache line to write buffer, flush line from write buffer, **Store** data uncached
Loads: **ldb, ldh, ldw,** **ldb.d, ldh.d, ldw.d**	Uncached write combine	Same as uncached.				

Table 23. Cache policy (continued)

Instruction	Policy	Cache	Write buffer	Prefetch	SCU	Result
Stores: **stb, sth, stw**	Uncached write combining	Miss	Miss	Miss		**Store** to write buffer
				Hit		Discard prefetch entry, **Store** to write buffer
			Hit			**Store** to write buffer
		Hit clean				Invalidate cache line, **Store** to write buffer
		Hit dirty				Purge cache line to write buffer, flush line from write buffer, **Store** data to write buffer
Loads: **ldb, ldh, ldw,**	Cached	Miss	Miss	Miss		Fill cache line from RAM, **Load** data from cache
				Hit		Transfer data from prefetch cache to data cache, **Load** data from data cache.
			Hit			Flush write buffer line, Fill cache line, **Load** data from cache
		Hit clean				**Load** data from cache
		Hit dirty				**Load** data from cache
Dimissable Loads: **ldb.d, ldh.d, ldw.d**	Cached	Miss	Miss	Miss	Hit	Fill cache line from ram, **Load** data from cache
					Miss	Return 0
				Hit		Transfer data from prefetch cache to data cache, **Load** data from data cache.
			Hit		Hit	Flush write buffer line, fill cache line, **Load** data from cache
					Miss	Flush write buffer line, Return 0.
		Hit clean				**Load** data from cache
		Hit dirty				**Load** data from cache
Stores: **stb, sth, stw**	Cached	Miss	Miss	Miss		**Store** data to write buffer
				Hit		Invalidate prefetch cache line, **Store** data to write buffer
			Hit			**Store** data to write buffer
		Hit clean				**Store** data to cache & dirty cache line
		Hit dirty				**Store** data to cache

Table 23. Cache policy (continued)

Instruction	Policy	Cache	Write buffer	Prefetch	SCU	Result
prgadd	All	Miss	Miss	Miss		No effect
				Hit		Invalidate prefetch cache line
			Hit			No effect
		Hit clean				Invalidate cache line
		Hit dirty				Purge cache line to write buffer
prgset	All					Purge cache lines in set (dirty data to write buffer) and invalidate entire prefetch cache.
sync						Flush entire write buffer to memory, wait for all outstanding writes, prefetches and uncached writes to complete.
pft	Cached	Miss	Miss	Miss	Hit	Prefetch sent to prefetch cache
					Miss	Prefetch discarded *(may use prefetch cache slot until SCU is checked)*
				Hit		Prefetch discarded
			Hit			Prefetch discarded
		Hit				Prefetch discarded
	Uncached					Prefetch discarded
	Uncached write combining					Prefetch discarded

7.3.12 Write buffer

Stores that miss the data cache and dirty lines that are evicted from the cache are held in the write buffer, pending write back to external memory.

The write buffer is a write combining buffer that holds up to four entries. Each entry has 32 bytes of data, an address and 32 bits of byte masks. The write buffer is operated as an LRU (least recently used) buffer.

Write combining allows individual close proximity writes to be merged into a single bus write. Write combining improves performance significantly (for a no write allocate cache) when performing sequences of writes to blocks of data which have not been brought into the cache.

7.4 Core memory controller (CMC)

The CMC allows multiple masters to access the STBus though a single port. The CMC arbitrates between multiple requestors and correctly routes responses.

7.5 Additional notes

The memory subsystem requires some additional explanation of some key operations and methods of use. This section is intended to provide this information without filling out the previous sections.

7.5.1 Memory ordering and synchronization

Use a **sync** operation to enforce the completion and ordering of memory operations.

A **sync** operation ensures that:

- the write buffer is empty (by flushing it to memory)
- all outstanding writes have reached external memory (both cached and uncached)
- all outstanding prefetches have completed

After any purge operations have taken place a **sync** should be issued to ensure that dirty data that was sent to the write buffer is flushed to memory.

7.5.2 Coherency between I-side and D-side

There is no coherency guaranteed between the external memory and the D-side and I-side memory subsystems. If coherency is desired then the memory subsystem has to be purged and synchronized.

This is achieved by the following sequence.

1. Flush the entire data cache by issuing **prgset** or **prgadd** operations.
2. Ensure all data is written back to memory by issuing a **sync**.
3. Invalidate the instruction cache by issuing **prgins** or **prginspg** operations.
4. Flush the instruction buffer by issuing a **syncins**.

7.5.3 Reset state

After reset all lines in the instruction cache and data cache are marked as invalid. The write buffer and prefetch cache entries are marked as empty.

7.5.4 Cached data in uncached region

If data from an uncached region is in the cache, then accessing the data as uncached causes it to be purged from the cache. This ensures that uncached accesses are always performed directly on the memory.

7.5.5 Prefetch performance

The prefetch cache is intended to improve performance. This can be achieved by explicitly fetching data from external memory and hiding the associated latency. The following points must be considered to ensure the prefetch cache works effectively.

- Data must be prefetched well in advance of use. The latency of an external memory access needs to be hidden between the **pft** operation and the first **load** operation which uses data from the prefetched line. This latency is in the region of 80-120 cycles for stall free bundles.
- The prefetch cache size should be taken into account, such that the number of outstanding prefetches does not exceed the number of entries in the prefetch cache.
- Unused prefetches increase bandwidth and waste entries in the prefetch cache.

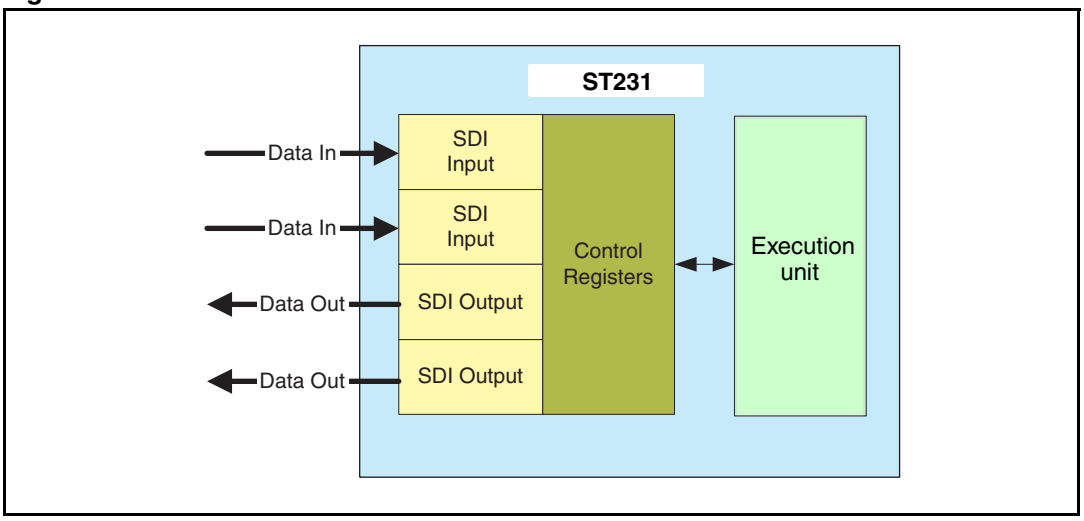
The first two points indicate a window for which prefetches might be considered.

8 Streaming data interface

The ST231 streaming data interface (SDI) is designed to allow fast and easy connection of on-chip peripherals. Each SDI is unidirectional and includes handshakes to prevent data loss and improve data flow.

The ST231 implements four SDI interfaces, two input ports and two output ports.

Figure 9. SDI overview



The SDIs:

- provide a mechanism for attaching streaming hardware to the processor core
- reduce STBus traffic and associated processor stall cycles
- reduce cache pollution and control complexity
- prevent deadlock through a timeout mechanism
- allow communication between clock domains without complex synchronization hardware

The SDI ports are accessed using control registers in the core.

8.1 Functional description

Data is communicated through either an output port or an input port to the processor. Data is communicated in order, that is, the n^{th} data item communicated arrives after the $(n - 1)^{th}$ item and before the $(n + 1)^{th}$ data item.

The SDI blocks writes (stores) to an output port if the SDI is full. Conversely, it blocks reads (loads) from an input port if the channel is empty. The ST231 blocks execution by stalling the entire processor. No execution proceeds until the channel becomes ready for the requested communication or an interrupt or timeout exception occurs.

8.1.1 Data width

The SDI port data interface is 32-bits wide. External to the SDI port, however, the data width can be arbitrary. For example connecting to a DCT peripheral which consumes 16-bits, data could be sent from the ST231 as single 16-bit items. The ST231 can only write 32-bit data to control registers, so writing a pair of 16 bit values in a packed word would be twice as fast.

Note: In this case the peripheral has to expect pairs of 16-bit values.

8.2 Communication channel

In its basic form the SDI acts as a communication channel to and from the processor. It is fully synchronized, allowing idealized input and output to be dealt with directly by the processor using load and store operations directly access the processor registers.

The SDI accesses can be initiated from C program code as accesses to volatile variables.

8.2.1 Timeouts

The timeouts operate as monitors to each individual SDI access. If an access remains stalled for too long, as defined by the control registers, an exception occurs.

8.3 Registers

The SDI interfaces directly to the ST231 load store unit. The interface is through a number of memory mapped registers in the control register address space.

The addresses of these registers are provided in *Chapter 9: Control registers on page 67*.

8.3.1 Input channel memory mapping

SDI*i*_DATA — The SDI*i*_DATA register is the location from which data is read from the input channel. The processor control and channel logic synchronize to ensure no data is lost. If the SDI*i*_DATA register is empty the processor stalls. Writing this register has no effect and the processor does not stall.

SDI*i*_READY — The SDI*i*_READY register is implementation specific. If non zero it indicates that the channel has data ready to be read.

This value indicates a minimum number of ready items. Returning the exact amount of data ready to be read from the channel may not be possible for a number of reasons, that is, clock boundary issues, propagation delays, hence the looser condition of the minimum number of ready items. In its simplest form this ready value can be 1, indicating at least one item is ready.

SDI*i*_CONTROL — The SDI*i*_CONTROL register is used to reset the channel and the SDI*i*_TIMEOUT register. The usage of the bits are defined in *Table 24*. The definition of the privilege bits is given in *Section 8.3.3: Protection on page 64*.

SDI*i*_COUNT — The SDI*i*_TIMEOUT register is reset to this value each time an SDI data value is successfully accessed. The value may be read or written. At reset it is set to a fixed value defined by the particular implementation. Time-outs can be disabled via the SDI*i*_CONTROL register.

SDI*i*_TIMEOUT — The number of cycles an SDI data access is allowed to stall before a timeout exception is thrown. The value may be read or written. This register is normally set to the value of SDI*i*_COUNT. Exceptions to this are when it has been specifically set to another value, or when an SDI access has taken a timeout exception or been interrupted. In the case of an SDI timeout the SDI*i*_TIMEOUT register contains the value zero.

Table 24. SDI0_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
PRIV	[1:0]	RW	0x0	Privilege bits.
RESETINPUT	2	RO	0x0	RESETINPUT (read only) acts as RESETREQUEST when slave, RESETACK when master.
RESETOUTPUT	3	RW	0x0	RESETOUTPUT, acts as RESETREQUEST when master, RESETACK when slave.
INPUTNOTOUTPUT	4	RO	0x0	INPUTNOTOUTPUT (read only).
Reserved	5	RO	0x0	Reserved
MASTERNOTSLAVE	6	RO	0x0	MASTERNOTSLAVE (read only).
TIMEOUTENABLE	7	RW	0x0	Timeout disable (set to 1 to disable timeout interrupts).
Reserved	[31:8]	RO	0x0	Reserved

8.3.2 Output channel memory mapping

SDI*i*_DATA — The SDI*i*_DATA register is the location from which data is written to the output channel. The processor control and channel logic synchronize to ensure no data is overwritten. If the SDI*i*_DATA register is full, the processor stalls. Reading this value has no effect and the processor does not stall and a value of zero is returned.

SDI*i*_READY — The SDI*i*_READY register is implementation specific. If non zero it indicates that the channel has space where data can be written.

This value indicates a minimum number of empty spaces where data can be written. In an implementation where the channel is connected to a FIFO this register could indicate, full, not full, the FIFO is half empty by returning (for example) the values 0, 1, 32. Returning the exact amount of data space available in the channel may not be possible for a number of reasons, that is, clock boundary issues, propagation delays, hence the looser condition of the minimum number of ready items. In the simplest form this ready value can be 1, indicating at least one item can be written.

SDI*i*_CONTROL — Bits defined as *Section 8.3.1: Input channel memory mapping on page 63*.

SDI*i*_COUNT — Defined as *Section 8.3.1: Input channel memory mapping on page 63*.

SDI*i*_TIMEOUT — Defined as *Section 8.3.1: Input channel memory mapping on page 63*.

8.3.3 Protection

The SDI register space is protected from malicious usage via access permissions held in each SDIi_CONTROL register. The reset behavior is that accesses to the SDI registers are only allowed in supervisor mode.

The protection can be loosened to allow user access to an SDIi_DATA and SDIi_READY registers. This is achieved via the SDIi_CONTROL register, PRIV[1:0] two bit field, indicating the access allowed for each SDI.

Table 25 lists the SDI_CONTROL_PRIV values.

Table 25. SDI_CONTROL_PRIV values

Name	Value	Comment
PRIV_NOUSER	0	Access only allowed in supervisor mode
PRIV_USER	1	Allow user access to data and ready register
Reserved	2	Reserved (defaults to privilege no user).
Reserved	3	Reserved (defaults to privilege no user).

8.4 Interrupts, exceptions and restarts

This section provides information about:

- interrupts, including returns from an interrupt and accesses to SDI registers
- SDI exceptions
- restarts or soft resets

8.4.1 Interrupts

The processor can take any interrupt while stalled accessing the SDI. An interrupt would be taken as if it occurred just prior to the bundle accessing the SDI.

Return from interrupt

The **rfi** from the exception handler continues program execution at the point prior to the interrupt.

If the SDI is ready by the completion of the interrupt handler, the SDI*i*_TIMEOUT register is reset to the value in SDI*i*_COUNT.

If, however, the SDI is still not ready upon completion of the handler, the processor reverts to the stalled state. The processor continues to wait for the channel to become ready while counting down the SDI*i*_TIMEOUT register from the value held prior to the exception.

Access to SDI registers

The interrupt handler can access all the SDI registers.

Note: Accessing the SDI*i*_DATA *register may alter the state of the processor observed by the interrupted processor.*

8.4.2 SDI exceptions

In this case the EXCAUSENO register indicates an SDI timeout exception. The exception address points to the SDI register on which the processor was waiting when the exception occurred.

A timeout exception occurs if the processor is actively waiting for a response from the interface for longer than the interface's timeout period.

The exception handler, in the case of a SDI timeout exception, is able to restart the communicating process. This is achieved by executing an **rfi** to the instruction that caused the exception. This causes re-execution of the instruction accessing the SDI.

Note: The SDI*i*_TIMEOUT *register must be increased from the zero value that caused the exception, otherwise the exception is triggered again immediately.*

The timeout exception is generated by the processor and not the channel.

8.4.3 Restart (soft reset)

A channel can only be restarted by the master of the channel.

A master may be either an input or an output channel. *Figure 10* shows how the reset structure is connected.

A reset is initiated by the master by setting the resetrequest bit in the control register. This causes the channel to begin the reset process. Once acknowledged (that is RESETACK = 1) as having been received by the slave, and consequently the entire channel structure being reset, the reset is removed by the software (that is, RESETREQUEST = 0) is communicated to the slave port. Once acknowledged (that is, RESTACK = 0), this indicates that the entire channel has exited the reset state.

When the SDI channel is reset, any data buffered by the core is discarded. This has the effect of making an output channel ready (as the output buffers are empty) or making an input channel not ready (as the input buffer is empty) until more data is received.

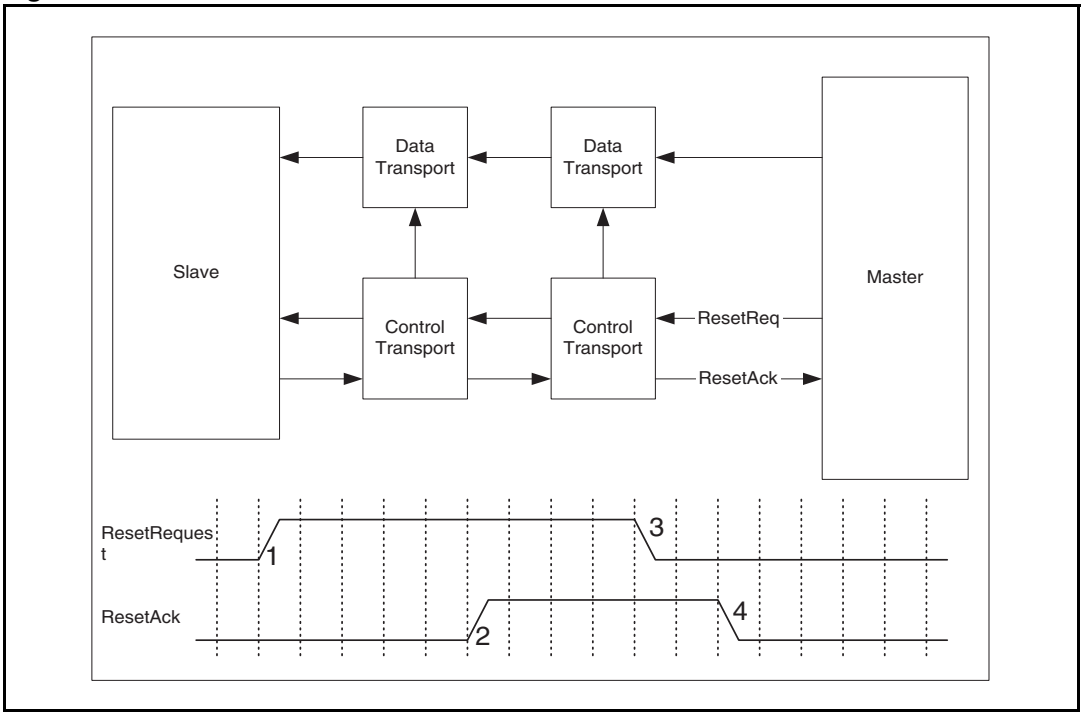
During the reset sequence the READY and DATA registers should not be accessed as the results are implementation dependant. The reset sequence does not effect the contents of the TIMEOUT and COUNT registers.

The slave reset can be used to reset a slave subsystem.

Normally the output channel is the master. However in cases where the output channel is connected to a dumb peripheral it may be necessary to make the input channel the master, particularly where this is a processor interface.

The restart structure outlined works across asynchronous clock boundaries. The reset control structure is shown in *Figure 10*.

Figure 10. Soft reset control structure



1. Master requests reset: Subsystem resets itself and consumes all data presented at inputs. RESETREQUEST is forwarded to other slave side subsystems.
2. All units in reset: After subsystem has reset itself AND all slave side subsystems have sent RESETACK, RESETACK can be forwarded to master.
3. Master requests leave reset: Unit forwards removal of RESETREQUEST to all slave-side subsystems. Unit leaves reset and stops consuming data.
4. All units out of reset: On receipt of RESETACK from all subsystems, RESETACK is forwarded to master. System can restart.

9 Control registers

The ST231 control registers contain processor state that is not typically accessed by application code. This includes accessing the TLB, PSW, exception registers and breakpoint registers.

9.1 Access operations

Control registers[a] are mapped into the address space, allowing access through normal load and store operations.

All control register accesses are word (32-bit) operations. Byte and half word loads and stores to control registers are not supported and generate CREG_ACCESS_VIOLATION exceptions.

Dismissible loads to control register space always return zero. Control register loads or stores are executed within the LSU without reference to the TLB regions.

9.2 Exceptions

The control register unit generates an exception when a **load** or **store** tries to:

- access a control register that does not exist (CREG_NO_MAPPING)
- access to a control register without correct permissions (CREG_ACCESS_VIOLATION)
- perform a byte or a half word access to control registers (CREG_ACCESS_VIOLATION)
- perform a misaligned word access to a control register (CREG_NO_MAPPING)

For details of the exception cause register, see *Section 5.6: Exception types and priorities on page 28*.

a. Control registers cannot be accessed from the STBus.

9.3 Control register addresses

Table 26 shows the addresses and access permissions of all control register addresses. The control registers are all relative to 0xFFFF 0000. Offsets listed in the table are relative to this, refer to *Section 6.2.2: Virtual addresses on page 32* for further information regarding virtual addresses. Control registers cannot be accessed from translated addresses, see *Section 6.5.5: Data accesses on page 45*.

The access column shows the access rights in user and supervisor mode:

NA No access (protection fault)

RO Read only, writes ignored.

ROF Read only fault on write.

RW Read/write.

CF Configurable read/write or no access.

CFRO Configurable read only or no access.

Table 26. Control registers - BASE: CREG_BASE

Name	Offset	Access (U/S)	Comment
PSW	0xFFF8	NA/RO	The program status word.
SAVED_PSW	0xFFF0	NA/RW	Saved PSW, written by hardware on exception.
SAVED_PC	0xFFE8	NA/RW	Saved program counter, written by hardware on exception.
HANDLER_PC	0xFFE0	NA/RW	The address of the exception handler code.
EXCAUSE	0xFFD8	NA/RO	A one hot vector of trap (exception/interrupt) types, indicating the cause of the last trap. Written by the hardware on a trap.
EXADDRESS	0xFFD0	NA/RW	This is the data effective address in the case of either a DPU, CREG, DBREAK, or MISALIGNED_TRAP exception. For other exception types this register is zero.
SAVED_SAVED_PSW	0xFFC0	NA/RW	PSW saved by debug unit interrupt.
SAVED_SAVED_PC	0xFFB8	NA/RW	PC saved by debug unit interrupt.
EXCAUSENO	0xFF88	NA/RW	Exception cause as an integer, indicating the cause of the last trap.
STATE1	0xFE00	NA/RW	Global machine state register. Controls cache locking.
VERSION	0xFFC8	NA/RO	The version number of the core.
PERIPHERAL_BASE	0xFFB0	NA/RO	Base address of peripheral registers. The top 12 bits of this register are wired to the peripheral base input pins.
SCRATCH1	0xFFA8	NA/RW	Scratch register reserved for use by supervisor software.

Table 26. Control registers - BASE: CREG_BASE (continued)

Name	Offset	Access (U/S)	Comment
SCRATCH2	0xFFA0	NA/RW	Scratch register reserved for use by supervisor software.
SCRATCH3	0xFF98	NA/RW	Scratch register reserved for use by supervisor software.
SCRATCH4	0xFF90	NA/RW	Scratch register reserved for use by the debug interrupt handler.
TLB_INDEX	0xFF80	NA/RW	Index of the TLB entry pointed to by TLB_ENTRY0-3.
TLB_ENTRY0	0xFF78	NA/RW	Bits [31:00] of the current TLB entry.
TLB_ENTRY1	0xFF70	NA/RW	Bits [63:32] of the current TLB entry.
TLB_ENTRY2	0xFF68	NA/RW	Bits [95:64] of the current TLB entry.
TLB_ENTRY3	0xFF60	NA/RW	Bits [127:96] of the current TLB entry.
TLB_EXCAUSE	0xFF58	NA/RW	Case of the TLB related exception.
TLB_CONTROL	0xFF50	NA/RW	Control bits for TLB.
TLB_REPLACE	0xFF48	NA/RW	Replacement pointer.
TLB_ASID	0xFF40	NA/RW	Current address space ID.
SCU_BASE0	0xD000	NA/RW	Base address of speculative load region.
SCU_LIMIT0	0xD008	NA/RW	Limit address of speculative load region.
SCU_BASE1	0xD010	NA/RW	Base address of speculative load region.
SCU_LIMIT1	0xD018	NA/RW	Limit address of speculative load region.
SCU_BASE2	0xD020	NA/RW	Base address of speculative load region.
SCU_LIMIT2	0xD028	NA/RW	Limit address of speculative load region.
SCU_BASE3	0xD030	NA/RW	Base address of speculative load region.
SCU_LIMIT3	0xD038	NA/RW	Limit address of speculative load region.
DBREAK_LOWER	0xFE80	NA/RW	Data breakpoint lower address.
DBREAK_UPPER	0xFE78	NA/RW	Data breakpoint upper address.
DBREAK_CONTROL	0xFE70	NA/RW	Data breakpoint control.
IBREAK_LOWER	0xFDD0	NA/RW	Instruction breakpoint lower address.
IBREAK_UPPER	0xFDC8	NA/RW	Instruction breakpoint upper address.
IBREAK_CONTROL	0xFDC0	NA/RW	Instruction breakpoint control.
PM_CR	0xF800	NA/RW	Performance monitoring control.
PM_CNT0	0xF808	NA/RW	Performance monitor counter 0 value.
PM_CNT1	0xF810	NA/RW	Performance monitor counter 1 value.
PM_CNT2	0xF818	NA/RW	Performance monitor counter 2 value.
PM_CNT3	0xF820	NA/RW	Performance monitor counter 3 value.
PM_PCLK	0xF828	NA/RW	Performance monitor core cycle counter.

Table 26. Control registers - BASE: CREG_BASE (continued)

Name	Offset	Access (U/S)	Comment
SDI0_DATA	0xE000	CF/RW	SDI 0 data.
SDI0_READY	0xE008	CFRO/RO	SDI 0 ready.
SDI0_CONTROL	0xE010	NA/RW	SDI 0 control.
SDI0_COUNT	0xE018	NA/RW	SDI 0 count.
SDI0_TIMEOUT	0xE020	NA/RW	SDI 0 timeout.
SDI1_DATA	0xE400	CF/RW	SDI 1 data.
SDI1_READY	0xE408	CFRO/RO	SDI 1 ready.
SDI1_CONTROL	0xE410	NA/RW	SDI 1 control.
SDI1_COUNT	0xE418	NA/RW	SDI 1 count.
SDI1_TIMEOUT	0xE420	NA/RW	SDI 1 timeout.
SDI2_DATA	0xE800	CF/RW	SDI 2 data.
SDI2_READY	0xE808	CFRO/RO	SDI 2 ready.
SDI2_CONTROL	0xE810	NA/RW	SDI 2 control.
SDI2_COUNT	0xE818	NA/RW	SDI 2 count.
SDI2_TIMEOUT	0xE820	NA/RW	SDI 2 timeout.
SDI3_DATA	0xEC00	CF/RW	SDI 3 data.
SDI3_READY	0xEC08	CFRO/RO	SDI 3 ready.
SDI3_CONTROL	0xEC10	NA/RW	SDI 3 control.
SDI3_COUNT	0xEC18	NA/RW	SDI 3 count.
SDI3_TIMEOUT	0xEC20	NA/RW	SDI 3 timeout.
RESERVEDFF38	0xFF38	NA/RO	Reserved
RESERVEDFF30	0xFF30	NA/RO	Reserved
RESERVEDFF28	0xFF28	NA/RO	Reserved
RESERVEDFF20	0xFF20	NA/RO	Reserved
RESERVEDFF18	0xFF18	NA/RO	Reserved
RESERVEDFF10	0xFF10	NA/RO	Reserved
RESERVEDFF08	0xFF08	NA/RO	Reserved
RESERVEDFF00	0xFF00	NA/RO	Reserved
RESERVEDFE40	0xFE40	NA/RO	Reserved
RESERVEDFE38	0xFE38	NA/RO	Reserved
RESERVEDFE30	0xFE30	NA/RO	Reserved
RESERVEDFE28	0xFE28	NA/RO	Reserved
RESERVEDFE20	0xFE20	NA/RO	Reserved
RESERVEDFE18	0xFE18	NA/RO	Reserved

Table 26. Control registers - BASE: CREG_BASE (continued)

Name	Offset	Access (U/S)	Comment
RESERVEDFE10	0xFE10	NA/RO	Reserved
RESERVEDFE08	0xFE08	NA/RO	Reserved

9.4 Data cache replacement state register

The STATE1 register controls the global state of the data cache replacement logic. *Table 27* lists the fields in the STATE1 register.

Table 27. STATE1 bit fields

Name	Bit(s)	Writable	Reset	Comment
PARTITION	[1:0]	RW	0x0	Sets the maximum value for the round robin data cache replacement pointers as (3 - PARTITION). 00: Replace ways 0-3. 01: Replace ways 0-2. 10: Replace ways 0-1. 11: Replace way 0 only. A full data cache purge using **prgset** operations is required following the update of the field.
Reserved	[31:2]	RO	0x0	Reserved.

When the partition field is changed, the current data in the cache is not altered in any way. If the software wishes to force data out of a particular partition following a change to this register, the data must be purged from the cache in the normal way (**prgadd** or **prgset**).

For more details on cache partitioning see *Section 7.3.2: Data cache partitioning on page 52*.

9.5 Version register

The VERSION register contains three fields which uniquely identify a particular release of the core. *Table 28* lists the fields in the VERSION register.

Table 28. VERSION bit fields

Name	Bit(s)	Writable	Reset	Comment
PRODUCT_ID	[15:0]	RO	Refer to datasheet	Revision of the ST200 core specified by CORE_VERSION below.
CORE_VERSION	[23:16]	RO	0x05	ST200 core type. 0x05 refers to the ST231 core.
DSU_VERSION	[31:24]	RO	Refer to datasheet	Version of the debugging support unit.

10 Timers

The ST231 provides three timers. These are controlled by registers mapped into the ST231 memory space, see *Chapter 11: Peripheral addresses on page 76*.

10.1 Operation

For each of the three timers, the TIMECOUNT*i* register (where *i* = 0, 1, 2) is the current value of the timer. When a timer is enabled its TIMECOUNT*i* value is decremented on each timer tick until zero is reached. Upon the next tick, TIMECOUNT*i* is loaded with TIMECONST*i*, the STATUS bit in TIMECONTROL*i* register is then set.

The ENABLE bit in the TIMECONTROL*i* register controls the enabling of interrupts for each timer. The STATUS bit in the TIMECONTROL*i* register is AND'ed with the interrupt enable to produce the timer interrupt line. When a value of 1 is written to the STATUS bit it is cleared (and thus the interrupt is cleared).

Timer counting is enabled by the ENABLE bit in the TIMECONTROL*i* register. Counters are not reset when disabled, hence initial values can be written using the TIMECOUNT*i* registers.

The frequency of timer ticks is controlled by programming the TIMEDIVIDE register.

These registers are covered in more detail in the following subsections.

10.1.1 TIMEDIVIDEi

The TIMEDIVIDE register sets the number of bus clock cycles between each timer tick. This register can be programmed with values between 0 and 65535 (using the bottom 16 bits only). The divide value is equal to the value of this register plus one. This register is reset to zero (divide by 1). Writing this register sets the divide value and reading it returns the current divide value.

It is recommended that the boot code sets up the TIMEDIVIDE register so that timer ticks occur every 1µs.

The bit fields for the TIMEDIVIDE register are listed in *Table 29*.

Table 29. TIMEDEVIDE bit fields

Name	Bit(s)	Writable	Reset	Comment
DIVIDE	[15:0]	RW	0x0	Number of clock cycles required to decrement the timers +1. A value of 0 causes the timers to decrement on every clock cycle.
Reserved	[31:16]	RW	0x0	Reserved.

10.1.2 TIMECOUNT*i*

The TIMECOUNT*i* register returns the current value of the timer counter *i*.

Write to these registers to set initial values for the counters.

The bit fields of the TIMECOUNT*i* are listed in *Table 30*.

Table 30. TIMECOUNT*i* bit fields

Name	Bit(s)	Writable	Reset	Comment
COUNT	[31:0]	RW	0x0	Current value of timer counter.

10.1.3 TIMECONST*i*

The TIMECONST*i* register contains the value loaded into timer *i* when timer *i* reaches zero. If interrupts are enabled, the value of TIMECONST*i* defines the number of ticks between interrupts.

The bit fields of the TIMECONST*i* are listed in *Table 31*.

Table 31. TIMECONST*i* bit fields

Name	Bit(s)	Writable	Reset	Comment
CONST	[31:0]	RW	0x0	Value to be reloaded when timer reaches zero.

10.1.4 TIMECONTROL*i*

The TIMECONTROL*i* register enables the timer, enables timer interrupts and clears a timer interrupt.

The bit fields of the TIMECONTROL*i* are listed in *Table 32*.

Table 32. TIMECONTROL*i* bit fields

Name	Bit(s)	Writable	Reset	Comment
ENABLE	0	RW	0x0	Enable the timer
INTENABLE	1	RW	0x0	Enable the timer interrupt.
STATUS	2	RW	0x0	Status of the timer interrupt. When 1 a timer has expired. Writing a 0 to this bit has no effect. Writing a 1 to this bit clears it.
Reserved	[31:3]	RO	0x0	Reserved

10.2 Timer interrupts

The timer interrupt lines are connected to external interrupt bits 2:0, see *Section 12.3: Interrupt registers on page 81*.

10.3 Programming the timers

The TIMECONST*i* registers set the value to be reloaded into the corresponding timer. The value is loaded on the timer tick after a zero is reached, such that, the duration between timers reaching zero is (TIMECONST*i* + 1). For example, setting the value to 99 causes a reload and timer interrupt (if enabled) every 100 ticks.

11 Peripheral addresses

On the ST231 the interrupt controller, debug support registers (DSU), DSU ROM and the timers are memory mapped peripherals. Under normal usage these peripherals should, with the exception of the DSU ROM, be mapped in an uncacheable region in the TLB.

11.1 Access to peripheral registers

Peripheral registers are accessed through an STBus port. On the ST231 writes to addresses on the STBus are posted (the processor does not wait for them to complete before continuing execution). In order to guarantee that a write to a peripheral register has completed it is necessary to issue a **sync** operation.

Note: This is essential in order to guarantee that an interrupt handler has cleared an interrupt or disabled a timer before doing an **rfi***.*

The ST231 peripheral registers only support word **load** and **store** transactions, other accesses result in a bus error. The DSU ROM can be accessed by word loads and instruction cache fills, word stores are ignored and other accesses result in a bus error.

11.2 Peripheral addresses

The base address of the peripheral registers is the value of the PERIPHERAL_BASE register. See *Chapter 9: Control registers on page 67*.

The access columns in *Table 33* and *Table 34* list the access rights for the listed registers. The abbreviations used in this column are:

NA	No access (protection fault).
RO	Read only, writes ignored.
ROF	Read only fault on write.
RW	Read/write.
CF	Configurable read/write or no access.
CFRO	Configurable read only or no access.

11.2.1 Interrupt controller and timer registers

The interrupt controller and timer registers are all relative to PERIPHERAL_BASE. They are listed in *Table 33*.

Table 33. Interrupt controller - BASE: INTCR_BASE

Name	Offset	Access (U/S)	Reset	Comment
INTPENDING0	0x0000	RO	0x0	Interrupt pending bits 31:0.
INTPENDING1	0x0008	RO	0x0	Interrupt pending bits 63:32.
INTMASK0	0x0010	RW	0x0	Interrupt mask bits 31:0.
INTMASK1	0x0018	RW	0x0	Interrupt mask bits 63:32.
INTTEST0	0x0020	RW	0x0	Interrupt test register bits 31:0.
INTTEST1	0x0028	RW	0x0	Interrupt test register bits 63:32.
INTCLR0	0x0030	RW	0x0	Interrupt clear register bits 31:0.
INTCLR1	0x0038	RW	0x0	Interrupt clear register bits 63:32.
INTSET0	0x0040	RW	0x0	Interrupt set register bits 31:0.
INTSET1	0x0048	RW	0x0	Interrupt clear register bits 63:32.
INTMASKCLR0	0x0108	RW	0x0	Interrupt mask clear bits 31:0.
INTMASKCLR1	0x0110	RW	0x0	Interrupt mask clear bits 63:32.
INTMASKSET0	0x0118	RW	0x0	Interrupt mask set bits 31:0.
INTMASKSET1	0x0120	RW	0x0	Interrupt mask set bits 63:32.
TIMECONST0	0x0200	RW	0x0	Timer constant.
TIMECOUNT0	0x0208	RW	0x0	Timer counter.
TIMECONTROL0	0x0210	RW	0x0	Timer control
TIMECONST1	0x0218	RW	0x0	Timer constant.
TIMECOUNT1	0x0220	RW	0x0	Timer counter.
TIMECONTROL1	0x0228	RW	0x0	Timer control
TIMECONST2	0x0230	RW	0x0	Timer constant.
TIMECOUNT2	0x0238	RW	0x0	Timer counter.
TIMECONTROL2	0x0240	RW	0x0	Timer control.
TIMEDIVIDE	0x0248	RW	0x0	Timer divide.
RESERVED50	0x0050	RO	0x0	Reserved
RESERVED58	0x0058	RO	0x0	Reserved
RESERVED60	0x0060	RO	0x0	Reserved
RESERVED68	0x0068	RO	0x0	Reserved
RESERVED70	0x0070	RO	0x0	Reserved
RESERVED78	0x0078	RO	0x0	Reserved
RESERVED80	0x0080	RO	0x0	Reserved

Table 33. Interrupt controller - BASE: INTCR_BASE (continued)

Name	Offset	Access (U/S)	Reset	Comment
RESERVED88	0x0088	RO	0x0	Reserved
RESERVED90	0x0090	RO	0x0	Reserved
RESERVED98	0x0098	RO	0x0	Reserved
RESERVED100	0x0100	RO	0x0	Reserved

11.2.2 DSU registers

The debug support unit registers are all relative to PERIPHERAL_BASE. They are listed in *Table 34*.

Table 34. Debug support unit - BASE: DSU_BASE

Name	Offset	Access (U/S)	Reset	Comment
DSR0	0x0000	RO	See *Table 28 on page 72*.	DSU version.
DSR1	0x0008	RW	See *Table 54 on page 92*.	DSU status.
DSR2	0x0010	RW	0x0	DSU output.
DSR3	0x0018	RW	0x0	DSU communication.
DSR4	0x0020	RW	0x0	DSU communication.
DSR5	0x0028	RW	0x0	DSU communication.
DSR6	0x0030	RW	0x0	DSU communication.
DSR7	0x0038	RW	0x0	DSU communication.
DSR8	0x0040	RW	0x0	DSU communication.
DSR9	0x0048	RW	0x0	DSU communication.
DSR10	0x0050	RW	0x0	DSU communication.
DSR11	0x0058	RW	0x0	DSU communication.
DSR12	0x0060	RW	0x0	DSU communication.
DSR13	0x0068	RW	0x0	DSU communication.
DSR14	0x0070	RW	0x0	DSU communication.
DSR15	0x0078	RW	0x0	DSU communication.
DSR16	0x0080	RW	0x0	DSU communication.
DSR17	0x0088	RW	0x0	DSU communication.
DSR18	0x0090	RW	0x0	DSU communication.
DSR19	0x0098	RW	0x0	DSU communication.
DSR20	0x00a0	RW	0x0	DSU communication.
DSR21	0x00a8	RW	0x0	DSU communication.

Table 34. Debug support unit - BASE: DSU_BASE (continued)

Name	Offset	Access (U/S)	Reset	Comment
DSR22	0x00b0	RW	0x0	DSU communication.
DSR23	0x00b8	RW	0x0	DSU communication.
DSR24	0x00c0	RW	0x0	DSU communication.
DSR25	0x00c8	RW	0x0	DSU communication.
DSR26	0x00d0	RW	0x0	DSU communication.
DSR27	0x00d8	RW	0x0	DSU communication.
DSR28	0x00e0	RW	0x0	DSU communication.
DSR29	0x00e8	RW	0x0	DSU communication.
DSR30	0x00f0	RW	0x0	DSU communication.
DSR31	0x00f8	RW	0x0	DSU communication.

11.2.3 DSU ROM

The DSU ROM starts from PERIPHERAL_BASE + 0x4000. See *Chapter 13: Debugging support (TAPLink) on page 87*.

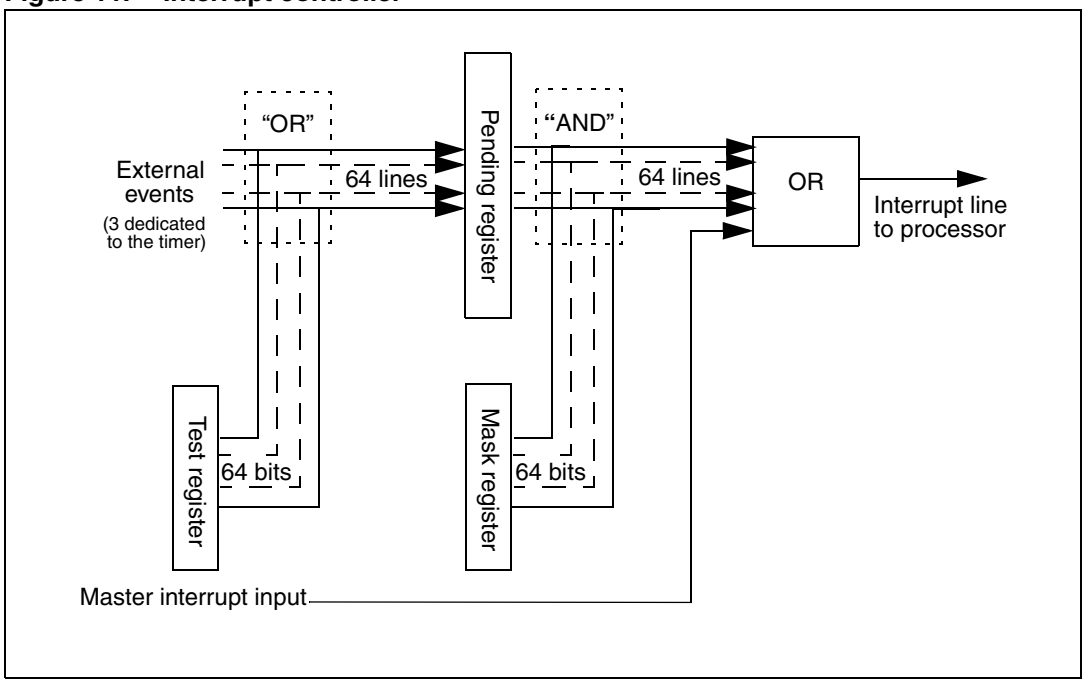
12 Interrupt controller

The ST231 interrupt controller supports up to 64 interrupt sources. The system is programmed using three pairs of 32-bit memory-mapped control registers.

12.1 Architecture

The structure of the interrupt controller is shown in *Figure 11*.

Figure 11. Interrupt controller



12.2 Operation

An interrupting event takes an interrupt line high. This is then sampled, causing the corresponding bit in the INTPENDING register to be set. The INTPENDING register is then parallel ANDed with the INTMASK register. The masked interrupts are then ORed into a single interrupt line that is presented to the processor core.

This architecture ensures that:

- all external interrupts interrupt the processor
- interrupts can be individually enabled or disabled

Setting or clearing bits in the INTMASK registers enables or disables the corresponding interrupt lines.

Note: The interrupt handling code is responsible for prioritization of interrupts. No hardware support is provided.

12.2.1 Test register

External interrupts are ORed with the contents of the INTTEST register before being sampled by the INTPENDING register. This allows the programmer to simulate interrupts into the processor for test purposes.

12.2.2 Master interrupt input

An external interrupt controller can be added to the core through the IRQ_MASTER_IN port. IRQ_MASTER_IN cannot be masked.

This is intended to be used instead of the internal interrupt controller. In this case, all interrupt inputs (IRQ) should be tied to zero, internal interrupts should be masked and timer interrupts outputs (IRQ_TIMER_OUT) should be connected to the external interrupt controller and dealt with through that.

12.3 Interrupt registers

For the addresses of these memory mapped registers see *Chapter 11: Peripheral addresses on page 76*.

12.3.1 Interrupt pending register (INTPENDING)

The INTPENDING registers are a pair of 32-bit registers that hold the current interrupt status. Bits in these registers are set by external interrupts or by the INTTEST registers.

Three bits in the INTPENDING registers are preassigned to the ST231 timer peripherals. The remaining 60 bits can be assigned to other peripherals or external devices.

Table 35 and *Table 37* list the bit fields in the INTPENDING registers, which indicate the pending interrupts.

INTPENDING0

Table 35. INTPENDING0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RO	0x0	Interrupt is pending from timer 0.
TIMER1	1	RO	0x0	Interrupt is pending from timer 1.
TIMER2	2	RO	0x0	Interrupt is pending from timer 2.
Reserved	[31:3]	RO	0x0	System defined interrupts 31:3 – refer to data sheet.

INTPENDING1

Table 36. INTPENDING1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	RO	0x0	System defined interrupts 63:32 – refer to data sheet.

12.3.2 Interrupt mask register (INTMASK)

The INTMASK registers are a pair of 32-bit registers whose contents are AND-ed with the corresponding INTPENDING register. They are used to enable and disable external interrupts.

Interrupts are enabled by setting, and disabled by clearing, the corresponding bits in the INTMASK registers.

Table 37 and *Table 38* list the bits of the INTMASK register.

INTMASK0

Table 37. INTMASK0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Mask bit for timer 0
TIMER1	1	RW	0x0	Mask bit for timer 1
TIMER2	2	RW	0x0	Mask bit for timer 2
Reserved	[31:3]	RW	0x0	Mask bits for system defined interrupts 31:3 – refer to data sheet.

INTMASK1

Table 38. INTMASK1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	RW	0x0	Mask bits for system defined interrupts 63:32 – refer to data sheet.

12.3.3 Interrupt mask set and clear registers (INTMASKSET and INTMASKCLR)

These registers provide a mechanism for atomically setting or clearing bits in the INTMASK registers, and remove the requirement for an uninterruptible Read-Modify-Write sequence.

When a program stores a 32-bit value into either of the INTMASKSET registers, any bits that are set to 1 cause the corresponding bit in the corresponding INTMASK register to be set to 1. Those bits that are 0 have no effect on the corresponding bits in the corresponding INTMASK register.

When a program stores a 32-bit value into either of the INTMASKCLR registers, any bits that are set to 1 cause the corresponding bit in the corresponding INTMASK register to be set to 0. Those bits that are 0 have no effect on the corresponding bits in the corresponding INTMASK register.

Table 39 shows the outcome of writing a value V to any of the four registers.

Table 39. Action of interrupt mask set and clear registers

Value "V" written to:	New value	
	INTMASK0	INTMASK1
INTMASKSET0	INTMASK0 **OR** V	INTMASK1
INTMASKSET1	INTMASK0	INTMASK1 **OR** V
INTMASKCLR0	INTMASK0 **AND** ~V	INTMASK1
INTMASKCLR1	INTMASK0	INTMASK1 **AND** ~V

OR is a bitwise OR, **AND** is a bitwise AND and ~V is the bitwise complement of V.

Table 40 and *Table 41* list the bits of the INTMASKCLR registers and *Table 42* and *Table 43* list the bits of the INTMASKSET registers.

INTMASKCLR0

Table 40. INTMASKCLR0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	WO	0x0	Mask clear bit for timer 0
TIMER1	1	WO	0x0	Mask clear bit for timer 1
TIMER2	2	WO	0x0	Mask clear bit for timer 2
Reserved	[31:3]	WO	0x0	Mask clear bits for system defined interrupts 31:3 – refer to data sheet.

INTMASKCLR1

Table 41. INTMASKCLR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	WO	0x0	Mask clear bits for system defined interrupts 63:32 – refer to data sheet.

INTMASKSET0

Table 42. INTMASKSET0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	WO	0x0	Mask set bit for timer 0
TIMER1	1	WO	0x0	Mask set bit for timer 1
TIMER2	2	WO	0x0	Mask set bit for timer 2
Reserved	[31:3]	WO	0x0	Mask set bits for system defined interrupts 31:3 – refer to data sheet.

INTMASKSET1

Table 43. INTMASKSET1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	WO	0x0	Mask set bits for system defined interrupts 63:32 – refer to data sheet.

12.3.4 Interrupt test register (INTTEST)

The INTTEST registers are a pair of 32-bit registers whose contents are ORed with the assertion state of external interrupts. It provides a mechanism for simulating interrupts to the processor.

Setting bits in the INTTEST registers causes the corresponding bits in the corresponding INTPENDING register to be set.

Table 44 and *Table 45* list the bits of the INTTEST register.

INTTEST0

Table 44. INTTEST0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Interrupt test bit for timer 0
TIMER1	1	RW	0x0	Interrupt test bit for timer 1
TIMER2	2	RW	0x0	Interrupt test bit for timer 2
Reserved	[31:3]	RW	0x0	Interrupt test bits for system defined interrupts 31:3 – refer to data sheet.

INTTEST1

Table 45. INTTEST1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	RW	0x0	Interrupt test bits for system defined interrupts 63:32 – refer to data sheet.

12.3.5 Interrupt set and clear registers (INTSET and INTCLR)

These registers provide a mechanism for atomically setting or clearing bits in the INTTEST registers, and remove the requirement for an uninterruptible Read-Modify-Write sequence.

When a program stores a 32-bit value into either of the INTSET registers, any bits that are set to 1 cause the corresponding bit in the corresponding INTTEST register to be set to 1. Those bits that are 0 have no effect on the corresponding bits in the corresponding INTTEST register.

When a program stores a 32-bit value into the INTCLR register, any bits that are set to 1 cause the corresponding bit in the corresponding INTTEST register to be set to 0. Those bits that are 0 have no effect on the corresponding bits in the corresponding INTTEST register.

Table 46 shows the outcome of writing a value V to any of the four registers.

Table 46. Action of interrupt set and clear registers

Value "V" written to:	New value	
	INTTEST0	INTTEST1
INTSET0	INTTEST0 **OR** V	INTTEST1
INTSET1	INTTEST0	INTTEST1 **OR** V
INTCLR0	INTTEST0 **AND** ~V	INTTEST1
INTCLR1	INTTEST0	INTTEST1 **AND** ~V

OR is a bitwise OR, **AND** is a bitwise AND and ~V is the bitwise complement of V.

Table 47 and *Table 48* list the bits of the INTCLR registers and *Table 49* and *Table 50* list the bits of the INTSET registers.

INTCLR0

Table 47. INTCLR0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	WO	0x0	Interrupt clear bit for timer 0
TIMER1	1	WO	0x0	Interrupt clear bit for timer 1
TIMER2	2	WO	0x0	Interrupt clear bit for timer 2
Reserved	[31:3]	WO	0x0	Interrupt clear bits for system defined interrupts 31:3 – refer to data sheet.

INTCLR1

Table 48. INTCLR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	WO	0x0	Interrupt clear bits for system defined interrupts 63:32 – refer to data sheet.

INTSET0

Table 49. INTSET0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	WO	0x0	Interrupt set bit for timer 0
TIMER1	1	WO	0x0	Interrupt set bit for timer 1
TIMER2	2	WO	0x0	Interrupt set bit for timer 2
Reserved	[31:3]	WO	0x0	Interrupt set bits for system defined interrupts 31:3 – refer to data sheet.

INTSET1

Table 50. INTSET1 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	WO	0x0	Interrupt set bits for system defined interrupts 63:32 – refer to data sheet.

13 Debugging support (TAPLink)

Note: The debugging support specified in this and the next chapter is implementation dependant. Check the product datasheet for details as to which support is provided.

Debugging support on the ST231 is provided by 4 main components.

Core	The ST231 core includes a non-maskable debug interrupt, and additional state to support the taking of debug interrupts. The core also contains hardware breakpoint support.
DSU	Shared DSU registers and state machine which generates debug interrupts and send responses over debug interface.
Debug ROM	Default program run in response to debug interrupt. This program uses the DSU registers to send higher level protocols over the debug interface. This program implements the **dsu_peek**, **dsu_poke**, **dsu_call_or_return** and **dsu_flush** operations.
Host debug interface	The hardware link, using the TAPLink protocol, to any connected host target interface (HTI). Supports **peek**, **poke**, **peeked** and **event** messages.

13.1 Core

This section describes debug interrupts, including entering and exiting debug mode, and hardware breakpoint support.

13.1.1 Debug interrupts

The ST231 can accept and service interrupts from the DSU. Debug interrupts are higher priority than normal interrupts, cannot be masked, and place the ST231 in a debug state.

A debug interrupt can be triggered either by an event from the host (see *Generating debug interrupts on page 97*) or by an external trigger, see *DSU status register (DSR1) on page 91*.

Entering debug mode

The ST231 handles a debug interrupt differently to other external interrupts. When a debug interrupt is taken, the TLB is disabled and the processor jumps to the start of the debug ROM. As the TLB is disabled, memory accesses above 0xFFFF 0000 access control registers (see *Section 6.5.5: Data accesses on page 45*). If access to physical memory from 0xFFFF 0000 upwards is required, the TLB must be re-enabled.

Taking a debug interrupt can be summarized as:

```
NEXT_PC ← DEBUG_HANDLER_PC;     // Branch to handler

SAVED_SAVED_PSW ← SAVED_PSW;    // Save the SAVED_PSW and
SAVED_SAVED_PC ← SAVED_PC;      // SAVED_PC

SAVED_PSW ← PSW;                // Save the PSW and PC
SAVED_PC ← BUNDLE_PC;           //

PSW[USER_MODE] ← 0;             // Enter supervisor mode
PSW[INT_ENABLE] ← 0;            // Disable interrupts
PSW[TLB_ENABLE] ← 0;            // Disables the TLB
PSW[DEBUG_MODE] ← 1;            // Enter debug mode
PSW[DBREAK_ENABLE] ← 0;         // Disable DBreak
PSW[IBREAK_ENABLE] ← 0;            // Disable IBreak
```

The EXCAUSENO and EXADDRESS registers are not updated when entering debug mode.

If the core accepts another debug interrupt whilst in debug mode, and if the debug interrupt is still pending when it leaves debug mode, the interrupt re-enters as normal.

The default debug ROM itself is not aware of virtual memory issues and provides the host with a physical view of memory. If address translation is in use, the operating system must install its own debug handler that is aware of virtual memory.

Exiting debug mode

When the DEBUG_MODE bit is cleared in the PSW, the ST231 exits debug mode and re-enters normal mode. If a debug interrupt is pending when the core leaves debug mode, it is re-entered as normal, see *Entering debug mode on page 88*.

Although clearing the DEBUG_MODE bit causes the ST231 to exit debug mode, attempting to set the DEBUG_MODE bit when it is not already set does not cause the core to enter debug mode and the DEBUG_MODE bit remains clear. The core can only enter debug mode by taking a debug interrupt from the DSU.

13.1.2 Hardware breakpoint support

Breakpoints are supported by:

- enable bits in the PSW
- address registers to define memory ranges
- control register to specify comparison operations

The safe way to use the breakpoint registers is to disable the breakpoints and then set the control and address registers before enabling the breakpoints again. This prevents spurious breaks due to inconsistent control and address registers.

Enable bits

Breakpoints are enabled though the PSW, one bit for instruction breakpoints and another data breakpoints, see *Section 3.4: Program status word (PSW) on page 19*.

Address registers

The ST231 uses two 32-bit registers to define addresses for the instruction and data breakpoints (IBREAK_LOWER, DBREAK_LOWER, IBREAK_UPPER, DBREAK_UPPER). These registers are all reset to the value 0.

Control registers

The IBREAK_CONTROL and DBREAK_CONTROL registers determine the comparison operations performed on the breakpoint addresses. If the comparison is true, then a breakpoint exception (IBREAK or DBREAK) is signaled.

For instruction breakpoints, the currently executing bundle address (PC) is used for comparison.

For data breakpoints, the data effective address of **load**s (both standard and dismissible) and **store**s are used for comparison. Prefetches and purges do not trigger data breakpoints.

Table 51 and *Table 52* provide details of the comparison operations defined for the instruction and data breakpoints.

Note: As the PC does not contain bits 1:0, these bits are ignored in any instruction address comparisons.

Table 51. DBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_IN_RANGE	0	RW	0x0	Break if address >= lower && address <=upper.
BRK_OUT_RANGE	1	RW	0x0	Break if address < lower || address > upper.
BRK_EITHER	2	RW	0x0	Break if address == lower || address == upper.
BRK_MASKED	3	RW	0x0	Break if address & upper == lower.
Reserved	[31:4]	RO	0x0	Reserved

Table 52. IBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_IN_RANGE	0	RW	0x0	Break if address >= lower && address <=upper.
BRK_OUT_RANGE	1	RW	0x0	Break if address < lower || address > upper.
BRK_EITHER	2	RW	0x0	Break if address == lower || address == upper.
BRK_MASKED	3	RW	0x0	Break if address & upper == lower.
Reserved	[31:4]	RO	0x0	Reserved

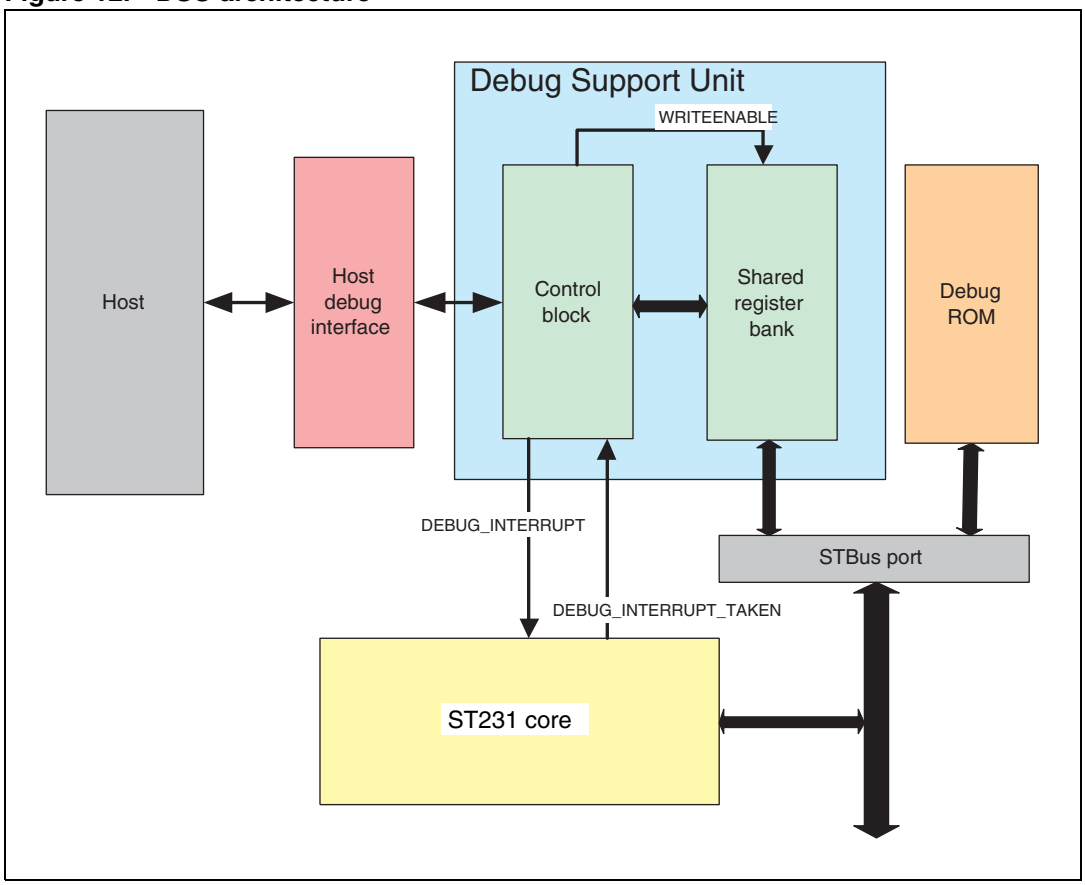
13.2 Debug support unit

The DSU allows both software and hardware to be debugged from a host by providing direct access to the ST231 core.

13.2.1 Architecture

Figure 12 shows the architecture of the DSU.

Figure 12. DSU architecture



The DSU is controlled by a host through the debug interface. The DSU control block interacts directly with the ST231 core through the DEBUG_INTERRUPT and DEBUG_INTERRUPT_TAKEN signals, and the shared register block.

The shared registers can also be accessed through the STBus port.

13.2.2 Shared register bank

The 32 shared registers consists of 3 reserved registers (DSR0-2) and 29 general purpose registers (DSR3-31). These are used to implement communication between the host and the target by the debug handler.

The shared register bank is 32-bits wide and only supports 32-bit STBus operations.

Table 53 lists the DSU shared registers.

Table 53. DSR_REG values

Name	Value	Comment
DSR0	0	DSU version register, contains version number for DSU, core and chip.
DSR1	1	DSU status register, contains DSU control and status bits.
DSR2	2	DSU output register, supports message transfer from target to HTI.
DSR3-31	3-31	General purpose registers.

The STBus addresses of the DSU registers are described in *Chapter 11: Peripheral addresses on page 76*.

Shared access conventions

The DSU shared registers are accessible independently from both the DSU and the STBus. The ST231 and DSU have no hardware support for synchronizing writes, so software conventions are used to prevent write conflicts.

13.2.3 DSU control registers

This subsection describes the DSU control registers: DSU version register, DSU status register and DSU output register.

DSU version register (DSR0)

The DSU version register is a read-only ID register. The fields are identical to the version register described in *Table 28: VERSION bit fields on page 72*.

DSU status register (DSR1)

The DSU status register contains the DSU status and control bits. See *Table 54*.

Table 54. DSR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
DEBUG_INTERRUPT_TAKEN	0	RO	0x0	Value of DEBUG_INTERRUPT_TAKEN signal, active high.
SUPERVISOR_WRITE_ENABLE	1	RW	0x1	STBus writes enabled if the core is in supervisor mode (regardless of debug mode).
USER_WRITE_ENABLE	2	RW	0x0	STBus writes enabled if the core is in user mode (regardless of debug mode).
BIGENDIAN	3	RO	0x0	When 1 the core is in big endian mode. When 0 the core is in little endian.
Reserved	4	RO	0x0	Reserved
OUTPUT_PENDING	5	RO	0x0	DSR2 contains a byte to be sent to the HTI which has not yet been sent.
TRIGGER_IN	6	RO	0x0	Current value of the trigger in pin.
TRIGGER_OUT	7	RW	0x0	Current value of the trigger out pin.
TRIGGER_ENABLE	8	RW	0x0	Enables/disables debug interrupts on trigger in.
Reserved	[15:9]	RO	0x0	Reserved
SW_FLAGS	[31:16]	RW	0x0	Reserved for future software use.

DSU output register (DSR2)

The lower 8 bits of the DSU output register are sent to the TAPLink (to any attached host) on being written. See *Table 55*.

Table 55. DSR2 bit fields

Name	Bit(s)	Writable	Reset	Comment
DATA	[7:0]	RW	0x0	Output data.
Reserved	[31:8]	RO	0x0	Always zero.

If OUTPUT_PENDING is non-zero then the byte most recently written has not yet been sent to the host target interface (HTI) and additional writes to the DSR2 do not affect the byte being sent even if they change the contents of the register.

Messages sent using the DSR2 may be delayed if the DSU is busy.

13.3 Debug ROM

The 1024-byte debug ROM is an ST231 peripheral. This contains the debug initialization loop and the default debug handler.

13.3.1 Debug initialization loop

On reset, the ST231 starts executing at the beginning of the boot ROM. However, if the DEBUG_ENABLE signal is asserted execution starts at the debug initialization loop (this is the first word of the debug ROM). This word contains a single syllable bundle which loops back to the same location, allowing the DSU to intervene and configure the core before it executes any code.

Note: Where the DEBUG_ENABLE *signal cannot be asserted, the boot ROM should start with a tight loop, or perhaps just a delay loop, to allow time for the DSU to interrupt the processor before it takes any action.*

13.3.2 Default debug handler

The default debug handler program starts at the second word of the debug ROM. It supports simple host-target debugging and the ability to install a more complex debug handler. The STBus address of the ROM is given in *Chapter 11: Peripheral addresses on page 76*.

Operation

On taking a debug interrupt, the default debug handler is executed. This first tests if a user handler is installed (that is, DSR3 is non zero) and if so branches to this address. The default debug handler then sends an event message to the host. This occurs even if DSU_COMMAND is 0. The handler then enters the command loop.

Command loop

The command loop reads and processes commands from a host, delivered via the TAPLink, to the DSU shared registers. Usage of the designated registers is shown in *Table 56*.

Table 56. Command register usage

Register name	Host use	Target use
DSU_COMMAND	Set with command	Zeroed when command accepted
DSU_ARG1,2,3	Set with arguments for command, before setting DSU_COMMAND	Set with response arguments before setting DSU_RESPONSE
DSU_RESPONSE	Zeroed after being read	Set to indicate outcome of a command

When the command is complete, the default debug handler stores the results in the argument registers and sets a success code in the response register.

Default handler commands

There are four default handler commands:

- `DSU_PEEK (DSU_COMMAND = 4)`

  Reads the 32-bit memory location addressed by DSU_ARG1 and returns the data in DSU_ARG1. The address must be word aligned. If the operation is successful DSU_RESPONSE is set to DSU_PEEKED (1) and a TAPLINK_EVENT_DEFAULT event is written to DSR2 causing an event with reason = 1 to be sent to the host.

Note: Any code greater than 4 is interpreted as a DSU_PEEK *command.*

- `DSU_POKE (DSU_COMMAND = 3)`

  Writes the 32-bit data word in DSU_ARG2 to the memory location addressed by DSU_ARG1. The address must be word aligned. If the operation is successful DSU_RESPONSE is set to DSU_POKED (2) and a TAPLINK_EVENT_DEFAULT event is written to DSR2 causing an event with reason = 1 to be sent to the host.

- `DSU_CALL_OR_RETURN (DSU_COMMAND = 1)`

  Calls the routine addressed by DSU_ARG1. If the called routine does not return this is effectively a branch. If DSU_ARG1 is zero this is a return call. Just before calling the user routine, or returning from a call, DSU_RESPONSE is set to DSU_RETURNING (3) and a TAPLINK_EVENT_DEFAULT event is written to DSR2 causing an event with reason = 1 to be sent to the host.

- `DSU_FLUSH (DSU_COMMAND = 2)`

  Flushes the address range starting at the value in DSU_ARG1 and ending at the value in DSU_ARG2 from data and instruction caches. If a command was successful DSU_RESPONSE is set to DSU_FLUSHED (4) and a TAPLINK_EVENT_DEFAULT event is written to DSR2 causing an event with reason = 1 to be sent to the host.

Trap handler

If a trap occurs while a command is being processed (for example, an invalid address is supplied on a **peek** or **poke**), the core deals with it as follows.

- The operation in progress is completed by loading the PC of the offending bundle, the exception cause number, and the exception address into DSR_ARG1, DSR_ARG2 and DSR_ARG3 respectively.
- DSU_RESPONSE is set to DSU_GOT_EXCEPTION (Code = 5) and a TAPLINK_EVENT_DEFAULT (Reason = 7) event is sent to the HTI.
- As with all exceptions, the SAVED_PC, SAVED_PSW, EXCAUSENO and EXADDRESS registers are updated when the exception occurs. The debug handler restores the values of these registers upon exit.

Context restore

Prior to exit the default handler restores any state it has altered.

Note: The context may have been further altered by commands issued.

Default handler register usage

The following DSU registers are defined and used by the default debug handler program:

Table 57. DSU command registers

DSR number	Designation	Comment
DSR3	DSR_USER_DEBUG_HANDLER	Control switches to this address if content is non-zero
DSR4-8(1)	DSU_ARG4-8	Not used in current debug handler
DSR9(1)	DSU_ARG3	Command argument 3
DSR10(1)	DSU_ARG2	Command argument 2. Used by DSU_POKE and DSU_FLUSH
DSR11(1)	DSU_ARG1	Command argument 1. Used by all DSU commands
DSR12	DSU_COMMAND	Command register. Written by HTI, cleared by target when command accepted
DSR13	DSU_RESPONSE	Response register. Set by target to a completion code, cleared by HTI before issuing next command
DSR14	Context saving	Saves SCR4_REG(2)
DSR15	Context saving	Saves SCR1_REG(2)
DSR16	Context saving	Saves SCR2_REG(2)
DSR17	Context saving	Saves SCR3_REG(2)
DSR18	Context saving	Saves the branch bits
DSR19	Context saving	Saves LINK_REG(2)
DSR20	Context saving	Saves HANDLER_PC
DSR21	Context saving	Saves SAVED_SAVED_PSW
DSR22	Context saving	Saves SAVED_SAVED_PC
DSR23	Context saving	Saves SAVED_PSW
DSR24	Context saving	Saves SAVED_PC
DSR25	Context saving	Saves EXCAUSENO
DSR26	Context saving	Saves EXADDRESS
DSR27-30	Unused	Unused
DSR31	Context saving	Saves DSR1

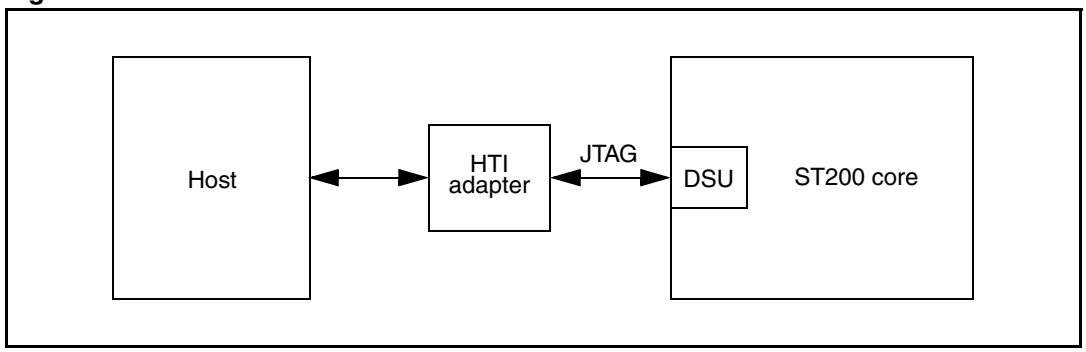
1. Argument registers are placed before the command register in the address space so that a command and its arguments can be loaded with a single poke operation.
2. As defined by the toolchain header files.

13.4 Host debug interface

The exchange of information with the host is through an HTI adapter. The DSU connects to the HTI using a JTAG interface, and the HTI connects to the host using Ethernet or USB. This is illustrated in *Figure 13*.

All host-target communication is carried out using **peek**, **poke**, **peeked** and **event** messages, passed between the host and the DSU.

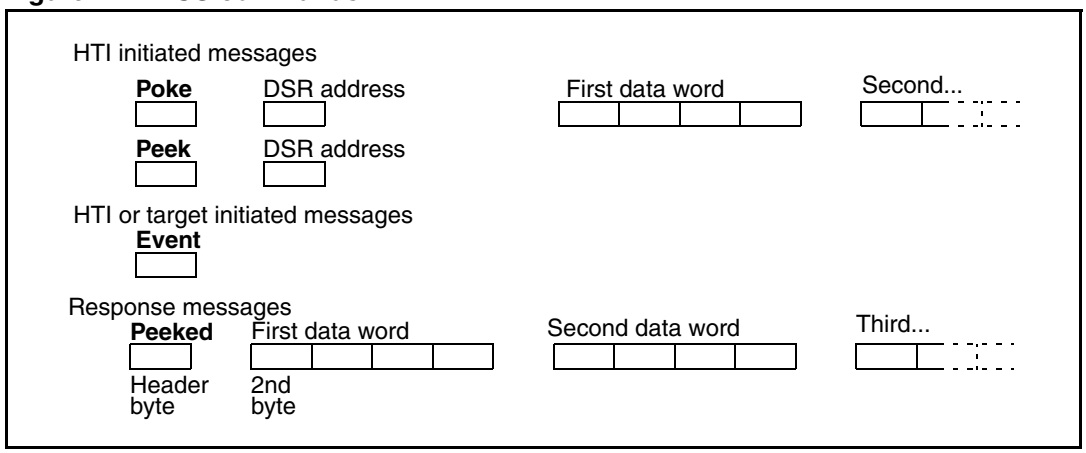
Figure 13. DSU overview



13.4.1 Message format

Commands are sent to the DSU in TAPLink message format consisting of a bidirectional byte stream which is interpreted by the DSU as a stream of commands. *Figure 14* shows the DSU commands in TAPLink message format.

Figure 14. DSU commands

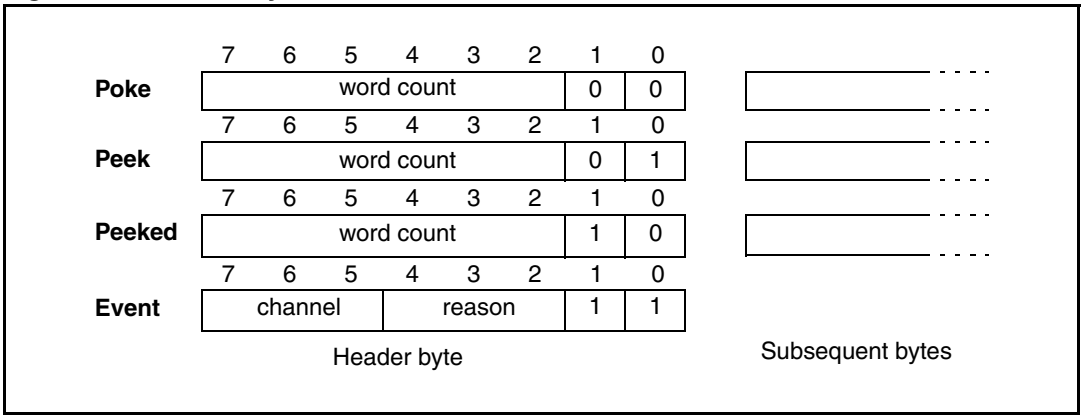


Note: Messages are transmitted little endian, irrespective of the endianness of the ST231 core.

Header bytes

Header bytes contain command-specific information such as the range of registers to be accessed. Header byte formats for the 4 DSU commands are illustrated in *Figure 15*.

Figure 15. Header bytes



Peek and poke operation

The **peek** and **poke** commands read and write the shared DSU registers. Each command uses a 6-bit count and one byte as a register address. The byte address references the first register in the range, and the count indicates how many registers are accessed. Counts greater than 32 have undefined results.

The result of a **peek** command is returned to the host using a PEEKED message.

Note: PEEKED messages are not supported from the host to the target and their behavior is undefined.

13.4.2 Operation

The operation of the host debug interface operation.

Generating debug interrupts

To interrupt the core, the host sends an EVENT with reason = 1. The EVENT is decoded by the DSU and a DEBUG_INTERRUPT signal is sent to the ST231. When the ST231 takes the interrupt (as described in *Section 13.1.1: Debug interrupts on page 87*), the DEBUG_INTERRUPT_TAKEN signal goes high.

The functionality available to the host depends upon the debug handler program running. The default handler uses designated shared registers to provide the higher level operations described in *Default handler commands on page 94*.

Core initiated events

The core can also request service from the host by sending it an EVENT message. This is done by writing the EVENT to the DSR2 (the output register). The channel and reason fields of the event message are not examined by the hardware and can be used as desired by the software.

14 Debugging support (JTAG)

Note: The debugging support specified in this and the previous chapter is implementation specific. Check the product datasheet for details as to which support is provided.

Debugging support on the ST231 is provided by 4 main components.

Core	The ST231 core includes a non maskable debug interrupt and additional state to support the taking of debug interrupts. The core also contains hardware breakpoint support.
DSU	Shared DSU registers and state machine which generates debug interrupts and send responses over debug interface.
Debug ROM	Default program run in response to debug interrupt. This program uses the DSU registers to send higher level protocols over the debug interface. This program implements the **dsu_peek**, **dsu_poke**, **dsu_call_or_return** and **dsu_flush** operations.
Host debug interface	The hardware link, using the JTAG protocol, to a host via a host target interface (HTI). Supports **peek**, **poke**, **peeked**, **poked**, **event**, **nop** and **event_ack** messages.

14.1 Core

This section describes debug interrupts, including entering and leaving debug mode, and hardware breakpoints.

14.1.1 Debug interrupts

The ST231 can accept and service interrupts from the DSU. Debug interrupts are higher priority than normal interrupts, cannot be masked, and place the ST231 in a debug state.

A debug interrupt can be triggered either by an event from the host (see *Host to DSU events on page 110*) or by an external trigger, see *DSU status register (DSR1) on page 102*.

Entering debug mode

A debug interrupt is handled differently to other external interrupts. When a debug interrupt is taken, the TLB is disabled and the processor jumps to the start of the debug ROM. As the TLB is disabled, memory accesses above `0xFFFF 0000` access control registers (see *Section 6.5.5: Data accesses on page 45*). If access to physical memory from `0xFFFF 0000` upwards is required, the TLB must be re-enabled.

Taking a debug interrupt can be summarized as:

```
NEXT_PC ← DEBUG_HANDLER_PC;     // Branch to handler

SAVED_SAVED_PSW ← SAVED_PSW;    // Save the SAVED_PSW and
SAVED_SAVED_PC ← SAVED_PC;      // SAVED_PC

SAVED_PSW ← PSW;                // Save the PSW and PC
SAVED_PC ← BUNDLE_PC;           //
```

```
PSW[USER_MODE] ← 0;          // Enter supervisor mode
PSW[INT_ENABLE] ← 0;         // Disable interrupts
PSW[TLB_ENABLE] ← 0;         // Disables the TLB
PSW[DEBUG_MODE] ← 1;         // Enter debug mode
PSW[DBREAK_ENABLE] ← 0;      // Disable DBreak
PSW[IBREAK_ENABLE] ← 0;         // Disable IBreak
```

The EXCAUSENO and EXADDRESS registers are not updated when entering debug mode.

If the core accepts another debug interrupt whilst in debug mode, and if the debug interrupt is still pending when it leaves debug mode, the interrupt re-enters as normal.

The default debug ROM itself is not aware of virtual memory issues and provides the host with a physical view of memory. If address translation is in use, the operating system must install its own debug handler that is aware of virtual memory.

Exiting debug mode

When the DEBUG_MODE bit is cleared in the PSW, the ST231 exits debug mode and re-enters normal mode. If a debug interrupt is pending when the core leaves debug mode, it is re-entered as normal, see *Entering debug mode on page 98*.

Although clearing the DEBUG_MODE bit causes the ST231 to exit debug mode, attempting to set the DEBUG_MODE bit when it is not already set does not cause the core to enter debug mode and the DEBUG_MODE bit remains clear. Debug mode can only be entered by taking a debug interrupt from the DSU.

14.1.2 Hardware breakpoint support

Breakpoints are supported by:

- enable bits in the PSW
- address registers to define memory ranges
- control register to specify comparison operations

The safe way to use the breakpoint registers is to disable the breakpoints and then set the control and address registers before enabling the breakpoints again. This prevents spurious breaks due to inconsistent control and address registers.

Enable bits

Breakpoints are enabled though the PSW, one bit for instruction breakpoints and another data breakpoints, see *Section 3.4: Program status word (PSW) on page 19*.

Address registers

The ST231 uses two 32-bit registers to define addresses for the instruction and data breakpoints (IBREAK_LOWER, DBREAK_LOWER, IBREAK_UPPER, DBREAK_UPPER). These registers are all reset to the value 0.

Control registers

The IBREAK_CONTROL and DBREAK_CONTROL registers determine the comparison operations performed on the breakpoint addresses. If the comparison is true, then a breakpoint exception (IBREAK or DBREAK) is signaled.

For instruction breakpoints, the currently executing bundle address (PC) is used for comparison.

For data breakpoints, the data effective address of loads (both standard and dismissible) and stores are used for comparison. Prefetches and purges do not trigger data breakpoints.

Table 58 and *Table 59* provide details of the comparison operations defined for the instruction and data breakpoints.

Note: As the PC does not contain bits 1:0, these bits are ignored in any instruction address comparisons.

Table 58. DBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_IN_RANGE	0	RW	0x0	Break if address >= lower && address <=upper.
BRK_OUT_RANGE	1	RW	0x0	Break if address < lower || address > upper.
BRK_EITHER	2	RW	0x0	Break if address == lower || address == upper.
BRK_MASKED	3	RW	0x0	Break if address & upper == lower.
Reserved	[31:4]	RO	0x0	Reserved

Table 59. IBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_IN_RANGE	0	RW	0x0	Break if address >= lower && address <=upper.
BRK_OUT_RANGE	1	RW	0x0	Break if address < lower || address > upper.
BRK_EITHER	2	RW	0x0	Break if address == lower || address == upper.
BRK_MASKED	3	RW	0x0	Break if address & upper == lower.
Reserved	[31:4]	RO	0x0	Reserved

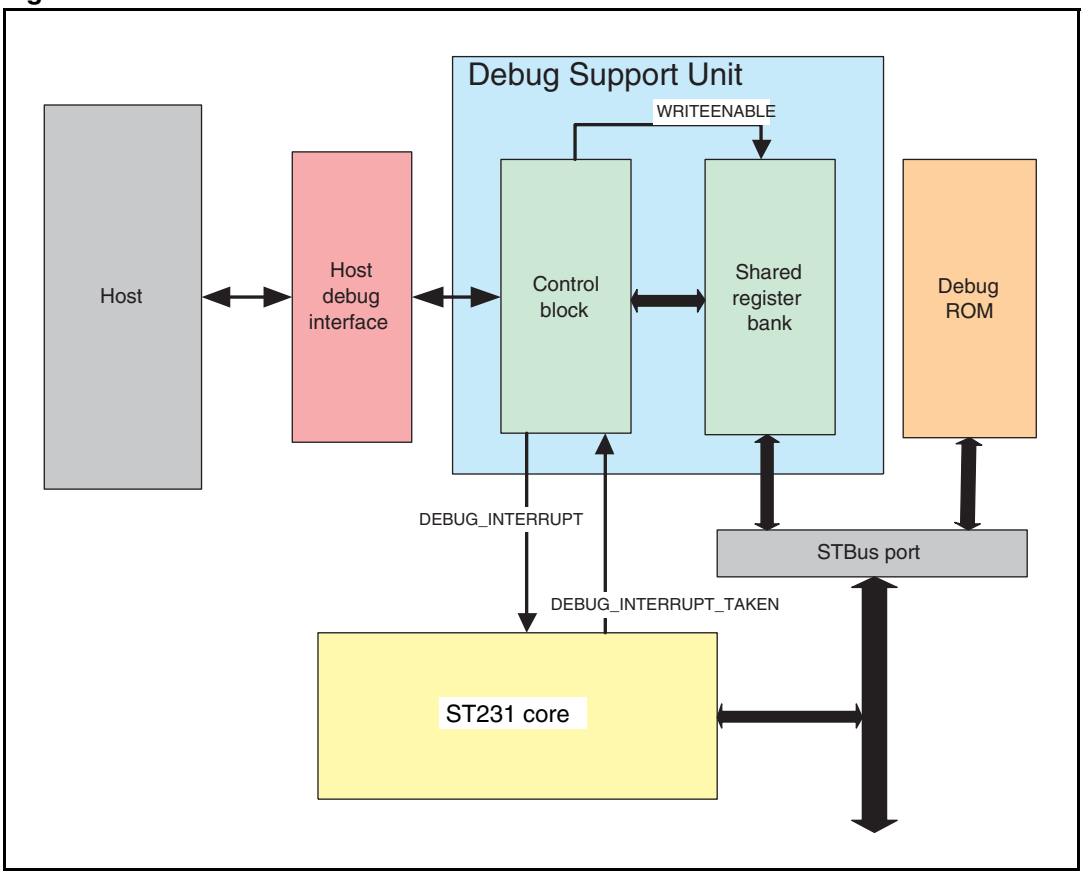
14.2 Debug support unit

The DSU allows both software and hardware to be debugged from a host by giving direct access to the ST231 core.

14.2.1 Architecture

Figure 16 shows the architecture of the DSU.

Figure 16. DSU architecture



The DSU is controlled by a host through the debug interface. The DSU control block interacts directly with the ST231 core through the DEBUG_INTERRUPT and DEBUG_INTERRUPT_TAKEN signals, and the shared register block.

The shared registers can also be accessed via the STBus port.

14.2.2 Shared register bank

The 32 shared registers consist of 3 reserved registers (DSR0-2) and 29 general purpose registers (DSR3-31). These are used to implement communication between the host and the target by the debug handler.

The shared register bank is 32-bits wide and only supports 32-bit STBus operations.

Table 60 lists the DSU shared registers.

Table 60. DSR_REG values

Name	Value	Comment
DSR0	0	DSU version register, contains version number for DSU, core and chip.
DSR1	1	DSU status register, contains DSU control and status bits.
DSR2	2	DSU output register, supports message transfer from target to HTI, see *Section 14.4 on page 107*.
DSR3-31	3-31	General purpose registers.

The STBus addresses of the DSU registers are described in *Chapter 11: Peripheral addresses on page 76*.

Shared access conventions

The DSU shared registers are accessible independently from both the DSU and the STBus. The ST231 and DSU have no hardware support for synchronizing writes, so software conventions are used to prevent write conflicts.

14.2.3 DSU control registers

This subsection describes the DSU control registers: the DSU version register, the DSU status register and the DSU output register.

DSU version register (DSR0)

The DSU version register is a read-only ID register. The fields are identical to the version register described in *Table 28: VERSION bit fields on page 72*.

DSU status register (DSR1)

The DSU status register contains the DSU status and control bits. See *Table 61*.

Table 61. DSR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
DEBUG_ INTERRUPT_ TAKEN	0	RO	0x0	Value of DEBUG_INTERRUPT_TAKEN signal, active high.
SUPERVISOR_ WRITE_ENABLE	1	RW	0x1	STBus writes enabled if the core is in supervisor mode (regardless of debug mode).

Table 61. DSR1 bit fields (continued)

Name	Bit(s)	Writable	Reset	Comment
USER_WRITE_ ENABLE	2	RW	0x0	STBus writes enabled if the core is in user mode (regardless of debug mode).
BIGENDIAN	3	RO	0x0	When 1 the core is in big endian mode. When 0 the core is in little endian.
HOST_EVENT_ ACK_PENDING	4	RW	0x0	The host is pending an event and an event_ack command is pending.
OUTPUT_PENDING	5	RO	0x0	DSR2 contains a byte to be sent to the HTI which has not yet been sent.
TRIGGER_IN	6	RO	0x0	Current value of the trigger in pin.
TRIGGER_OUT	7	RW	0x0	Current value of the trigger out pin.
TRIGGER_ENABLE	8	RW	0x0	Enables/disables debug interrupts on trigger in.
Reserved	[15:9]	RO	0x0	Reserved
SW_FLAGS	[31:16]	RW	0x0	Reserved for future software use.

DSU output register (DSR2)

A value written to DSR2 is sent to the host (using the HTI) by an EVENT message, which is handshaken with an EVENT_ACK message. See *DSU to host events on page 111* for details. *Table 62* gives details of the DSR2 register.

Table 62. DSR2 bit fields

Name	Bit(s)	Writable	Reset	Comment
DATA	[31:0]	RW	0x0	Output data.

14.3 Debug ROM

The 1024-byte debug ROM is an ST231 peripheral. This contains the debug initialization loop and the default debug handler.

14.3.1 Debug initialization loop

On reset, the ST231 starts executing at the beginning of the boot ROM. However, if the DEBUG_ENABLE signal is asserted execution starts at the debug initialization loop (this is the first word of the debug ROM). This word contains a single syllable bundle which loops back to the same location, allowing the DSU to intervene and configure the core before it executes any code.

Note: Where the DEBUG_ENABLE *signal cannot be asserted, the boot ROM should start with a tight loop, or perhaps just a delay loop, to allow time for the DSU to interrupt the processor before it takes any action.*

14.3.2 Default debug handler

The default debug handler program starts at the second word of the debug ROM. It supports simple host-target debugging and the ability to install a more complex debug handler. The STBus address of the ROM is given in *Chapter 11: Peripheral addresses on page 76*.

The value in SCRATCH4 is overwritten when the default debug handler starts and is not restored.

Operation

On taking a debug interrupt, the default debug handler is executed. This first tests if a user handler is installed (that is, DSR3 is non zero) and if so branches to the given address. See *Section 14.3.3: User-defined debug handler on page 107*. The default debug handler then sends an event message to the host. This occurs even if DSU_COMMAND is 0. The handler then enters the command loop.

Command loop

The command loop reads and processes commands from a host, delivered over the JTAG connection, to the DSU shared registers. Usage of the designated registers is shown in *Table 63*.

Table 63. Command register usage

Register name	Host use	Target use
DSU_COMMAND	The host sets this register to the command.	Zeroed when the command is accepted
DSU_ARG1,2,3	Set with arguments for the command, before setting DSU_COMMAND	Set with response arguments before setting DSU_RESPONSE
DSU_RESPONSE	After reading the value the host must write zero prior to sending the next command. See *Table 64 on page 106*.	Set to indicate outcome of a command

When the command is complete, the default debug handler stores the results in the argument registers and sets a success code in the response register.

Default handler commands

There are four default handler commands:

- `DSU_PEEK (DSU_COMMAND = 4)`

  Reads the 32-bit memory location addressed by DSU_ARG1 and returns the data in DSU_ARG1. The address must be word aligned. If the operation is successful DSU_RESPONSE is set to DSU_PEEKED (1).

  The value `0x7` is written to DSR2 causing an event with reason = 1 to be sent to the host, see *DSU to host events on page 111*.

Note: Any code greater than 4 is interpreted as a DSU_PEEK command.

- `DSU_POKE (DSU_COMMAND = 3)`

  Writes the 32-bit data word in DSU_ARG2 to the memory location addressed by DSU_ARG1. The address must be word aligned. If the operation is successful DSU_RESPONSE is set to DSU_POKED (2).

  The `0x7` is written to DSR2 causing an event with reason = 1 to be sent to the host, see *DSU to host events on page 111*.

- `DSU_CALL_OR_RETURN (DSU_COMMAND = 1)`

  Calls the routine addressed by DSU_ARG1. If the called routine does not return this is effectively a branch. If DSU_ARG1 is zero this is a return call. Just before calling the user routine, or returning from a call, DSU_RESPONSE is set to DSU_RETURNING (3).

  The `0x7` is written to DSR2 causing an event with reason = 1 to be sent to the host, see *DSU to host events on page 111*.

  The user routine may overwrite the following state without the need to save and restore: SCR1_REG, SCR2_REG, SCR3_REG, DSU_BASE_REG (see *Table 64*) and branch bit B0.

- `DSU_FLUSH (DSU_COMMAND = 2)`

  Flushes the address range starting at the value in DSU_ARG1 and ending at the value in DSU_ARG2 from data and instruction caches.

  If a command was successful DSU_RESPONSE is set to DSU_FLUSHED (4).

  The `0x7` is written to DSR2 causing an event with reason=1 to be sent to the host, see *DSU to host events on page 111*.

Trap handler

If a trap occurs while a command is being processed (for example, an invalid address is supplied on a **peek** or **poke**), the core deals with it as follows.

- The operation in progress is completed by loading the PC of the offending bundle, the exception cause number, and the exception address into DSU_ARG1, DSU_ARG2 and DSU_ARG3 respectively.
- DSU_RESPONSE is set to DSU_GOT_EXCEPTION (Code = 5).
- As with all exceptions, the SAVED_PC, SAVED_PSW, EXCAUSENO and EXADDRESS registers are updated when the exception occurred; the debug handler restores the values of these registers upon exit

Context restore

Before exiting the default handler restores any state it has altered.

Note: The context may have been further altered by commands issued.

Default handler register usage

The DSU registers in *Table 64* are defined and used by the default debug handler program.

Table 64. DSU command registers

DSR number	Designation	Comment
DSR3	DSR_USER_DEBUG_ HANDLER	Control switches to this address if content is non-zero
DSR4-8(1)	DSU_ARG4-8	Not used in current debug handler
DSR9(1)	DSU_ARG3	Command argument 3
DSR10(1)	DSU_ARG2	Command argument 2. Used by DSU_POKE and DSU_FLUSH
DSR11(1)	DSU_ARG1	Command argument 1. Used by all DSU commands
DSR12	DSU_COMMAND	Command register. Written by host, cleared by target when command accepted
DSR13	DSU_RESPONSE	Response register. Set by target to a completion code, cleared by host before issuing next command
DSR14	Context saving	Saves DSU_BASE_REG (R13)
DSR15	Context saving	Saves SCR1_REG (R9)
DSR16	Context saving	Saves SCR2_REG (R10)
DSR17	Context saving	Saves SCR3_REG (R11)
DSR18	Context saving	Saves branch bit B0
DSR19	Context saving	Saves LINK_REG(2)
DSR20	Context saving	Saves HANDLER_PC
DSR21	Context saving	Saves SAVED_SAVED_PSW
DSR22	Context saving	Saves SAVED_SAVED_PC
DSR23	Context saving	Saves SAVED_PSW
DSR24	Context saving	Saves SAVED_PC
DSR25	Context saving	Saves EXCAUSENO
DSR26	Context saving	Saves EXADDRESS
DSR27-30	Unused	Unused
DSR31	Context saving	Saves DSR1

1. Argument registers are placed before the command register in the address space so that a command and its arguments can be loaded with a single poke operation.

14.3.3 User-defined debug handler

A user-defined debug handler may be installed to replace the default debug handler. If DSR3 is non-zero, the core executes the following sequence to jump to the user-defined debug handler.

- SCR1_REG, SCR2_REG, SCR3_REG, DSU_BASE_REG and R63 are saved as shown in *Table 64*
- Branch bit B0 is saved as shown in *Table 64*
- SCRATCH4 is overwritten
- DSUBASE_REG is replaced by the base address of the DSU register block.
- A **goto** operation to the address in DSR3 is executed

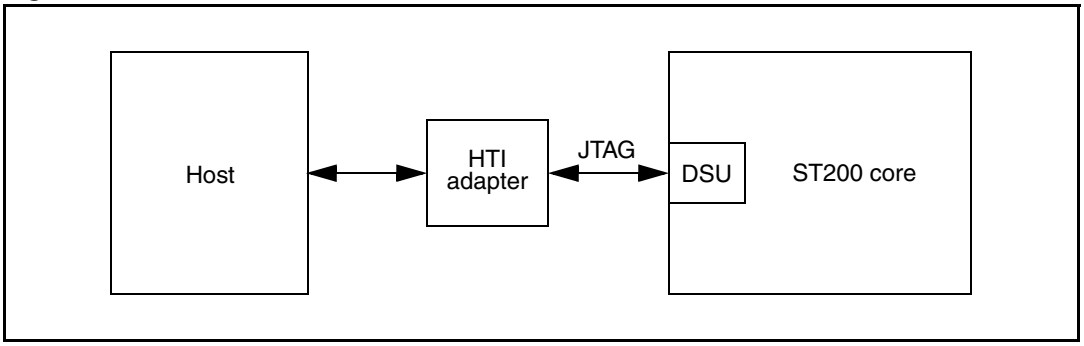
There is no facility for reusing the state restoration routine in the default debug handler for completion of a user-defined debug handler.

14.4 Host debug interface

Exchange of information with the host is through an HTI (host target interface) adapter. The DSU connects to the HTI through a JTAG interface, and the HTI connects to the host using Ethernet or USB. This is illustrated in *Figure 17*.

All host-target communication is done with **peek**, **poke**, **peeked, poked, nop, event and event_ack** commands sent between the host and the DSU.

Figure 17. DSU overview



The JTAG interface provides access to the registers within the DSU only, as described in *Section 14.2.2: Shared register bank on page 102*.

Access is made to memory using a software convention with the ST200 CPU as described in *Section 14.3.2: Default debug handler on page 104*.

event messages can be sent in either direction to allow software on the CPU to synchronize with software on the host.

14.4.1 Protocol and flow control

40-bit commands are exchanged between the host and the DSU using the JTAG port. Whenever a command is sent to the DSU by the host, the DSU responds with a response from a previous command, or a **nop** if no response is pending.

A symmetrical protocol is employed where every action request is handshaken. Therefore, the DSU sends the following responses:

- for a **peek** from the host, it sends a **peeked**
- for a **poke** from the host, it sends a **poked**
- for an **event** from the host, it sends an **event_ack**
- an **event_ack** from the host does not require a response; the DSU sends a **nop** if no other response is pending
- a **nop** from the host does not require a response; the DSU sends a **nop** if no other response is pending

The DSU sends a response to the i^{th} command, either after the DSU receives the $(i + 1)^{th}$ command (in the initial state), or after it receives the $(i + 2)^{th}$ command (in the buffered state).

In its initial state, the DSU responds to the i^{th} command when it receives the $(i + 1)^{th}$ command, and continues to do so until the processor writes to DSR2, sending an **event** to the host. The sending of the **event** is prioritized over the sending of a response to the i^{th} command, which is buffered.

In the state where there is a buffered response, DSU responds to the i^{th} command when it receives the $(i + 2)^{th}$ command. When the DSU receives an **event_ack** or a **nop** as a $(i + 2)^{th}$ command, it sends the response to the $(j + 1)^{th}$ command. Since neither the **event_ack** nor the **nop** require a response, the buffer is now empty, so the DSU re-enters the initial state.

As only one **event** can be outstanding to the host at a given time, the DSU is required to buffer one response only. As responses are not always sent immediately to incoming commands, the host must account for every **peek** and **poke** that is sent. The host must also poll the DSU with **nops** to receive **events**.

14.4.2 Command Format

Commands supported across the JTAG interface are as listed in *Table 65*. Commands are 40 bits long and consist of an 8-bit header and a 32-bit data field. The header is split into two fields. Commands are sent over the JTAG interface bit[0] first.

Table 65. JTAG commands

Command	header[2:0]	header[7:3]	data[39:8]	Command needs a response	Action / comment
Commands from the Host to the DSU					
nop	0x0	0x0	0x0	No	No action. No command is currently waiting to be sent. **nops** can be used to poll for **event**s.
peek	0x1	Address	0x0	Yes	Request to peek the DSU register specified by the address field. The DSU replies with **peeked**, **address**, **value**(1).
poke	0x2	Address	Data	Yes	Request to poke a DSU register specified by the address field with the value specified by the data field. The DSU replies with **poked**, **address**, **0**.
event	0x3	0x0	reason[10:8] channel[13:11] 0x0[39:14]	Yes	If reason = 1 and channel = 0 then raise a debug interrupt, otherwise a debug interrupt is not raised. The DSU replies with **event_ack**, **reason**, **channel**, **0**.
event_ack	0x4	0x0	DSR2[31:0]	No	An event from the DSU to the host has been processed. The original word in DSR2 is returned, but is not used.
reserved(2)	0x5-0x7	Undefined	Undefined	No	Reserved commands are treated as **nop**s.
Commands from the DSU to the host					
nop	0x0	0x0	0x0	No	No command is currently waiting to be sent.
peeked	0x1	Address	Value	No	Peeked data being returned to the host.
poked	0x2	Address	0x0	No	Response to a request to poke a DSU register.

Table 65. JTAG commands (continued)

Command	header[2:0]	header[7:3]	data[39:8]	Command needs a response	Action / comment
event	0x3	0x0	DSR2[31:0]	Yes	DSR2 [31:0] is copied into the data field, and the use is defined by software. Must be eventually replied to by an **event_ack**.
event_ack	0x4	0x0	reason[10:8] channel[13:11] 0x0[39:14]	No	An event has been processed (i.e. a debug interrupt has been applied to the core, it may not have been processed yet). The data field from the incoming command is placed in the data field of the response command.
reserved(2)	0x5-0x7	Undefined	Undefined	No	The behavior is defined by the host software.

1. The response may be delayed by one message if an **event_ack** is outstanding, as described in *Section 14.4.1 on page 108*.
2. Commands marked reserved are held for future development.

14.4.3 Handling events

This section describes how the DSU handles events.

Host to DSU events

The DSU generates an **event_ack** in response to an **event** command. The response indicates that the DSU has signalled a debug interrupt to the processor; it does not indicate that the processor has taken a debug interrupt (for instance, interrupts may have been disabled or the processor may be servicing a cache miss). The host can determine whether the processor is in debug mode by peeking DSR1. Bit 0 is set on entering debug mode and is cleared on exiting.

If the processor has not returned from debug mode, a subsequent event command causes an additional debug interrupt. The controlling software must ensure that events are not sent until previous events have been completely processed.

DSU to host events

Multiple events can be sent from the host to the DSU, but only one outstanding DSU-to-host event is permitted. Two bits in DSR1 give information about the current DSU-to-host event status as shown in *Table 66*.

Table 66. Status of events and DSR1 bit fields

OUTPUT_ PENDING DSR1[5]	HOST_EVENT_ACK_ PENDING DSR1[4]	Comment
0	0	No outstanding DSU to host **event**.
1	0	DSR2 has been written to, **event** has not been sent yet. Writes to DSR2 before DSR1[4] is set do not cause extra events, but update the value of DSR2 which is sent with the **event**.
1	1	This case does not occur. DSR1[5] and DSR1[4] are mutually exclusive.
0	1	The **event** has been sent. Writes to DSR2 do not cause further events to be sent.
0	0	The **event_ack** has been received. Writes to DSR2 cause **event** again.

15 Performance monitoring

The ST231 provides a hardware instrumentation system which consists of the following:

- a control register (PM_CR)
- a core clock counter (PM_PCLK)
- four event counters (PM_CNT***i***, ***i*** = 0, 1, 2, 3)

They are all mapped to addresses in the control register space as defined in *Section 9.3: Control register addresses on page 68*.

15.1 Events

The programmable events supported by the ST231 are listed in *Table 67*.

Table 67. PM_EVENT values

Name	Value	Comment
PM_EVENT_DHIT	0	Number of cached loads and stores that hit the cache.
PM_EVENT_DMISS	1	Number of cached loads and stores that miss the cache. This includes stores that miss the cache and are sent to the write buffer.
PM_EVENT_DMISSCYCLES	2	Number of cycles the core is stalled waiting for load/store operations to complete (this includes DTLB and uncached stalls).
PM_EVENT_PFTISSUED	3	Number of prefetches that are sent to the bus.
PM_EVENT_PFTHITS	4	Number of cached loads that hit the prefetch buffer.
PM_EVENT_WBHITS	5	Number of writes that hit the write buffer.
PM_EVENT_IHIT	6	Number of accesses the instruction buffer made that hit the instruction cache.
PM_EVENT_IMISS	7	Number of accesses the instruction buffer made that missed the instruction cache.
PM_EVENT_IMISSCYCLES	8	Number of cycles the instruction cache was stalled for due to refill from the STBus.
PM_EVENT_IBUFINVALID	9	Duration where IBuffer is not able to issue bundles to the pipeline.
PM_EVENT_BUNDLES	10	Bundles executed.
PM_EVENT_LDST	11	Load/Store instructions executed. These include: stw, sth, stb, pft, prgadd, prgset, prginspg, pswset, pswclr, sync, ldb, ldb.d, ldbu, ldbu.d, ldh, ldh.d, ldhu, ldhu.d, ldw, ldw.d.
PM_EVENT_TAKENBR	12	Number of taken branches, includes br, brf, rfi, goto and call.
PM_EVENT_NOTTAKENBR	13	Number of not taken branches (br and brf).
PM_EVENT_EXCEPTIONS	14	Number of exceptions and debug interrupts.

Table 67. PM_EVENT values (continued)

Name	Value	Comment
PM_EVENT_INTERRUPTS	15	Number of interrupts.
PM_EVENT_BUSREADS	16	Number of architectural read transactions issued to the bus. This is the number of uncached reads, I & D cache refills and prefetches issued to the bus.
PM_EVENT_BUSWRITES	17	Number of architectural write transactions issued to the bus. This is the number of write buffer lines evicted and the number of uncached writes issued to the bus.
PM_EVENT_OPERATIONS	18	Number of completed operations. Includes nops in the instruction stream but not those added dynamically. This counter excludes long immediates.
PM_EVENT_WBMISSES	19	Number of writes that missed the cache and missed the write buffer. This excludes cache line evictions.
PM_EVENT_NOPBUNDLES	20	Number of completed bundles that were empty or contained only nops. This includes nop bundles generated by instruction buffer stalls and interlocking stalls. It excludes pipeline stalls due to load/stores and control register/SDI accesses.
PM_EVENT_LONGIMM	21	Number of long immediates in completed bundles.
PM_EVENT_ITLBMISS	22	Number of instruction cache reads that missed the ITLB.
PM_EVENT_DTLBMISS	23	Number of load/store operations that missed the DTLB when the TLB is enabled.
PM_EVENT_UTLBHIT	24	Number of accesses to the UTLB which were hits.
PM_EVENT_ITLBWAITCYCLES	25	Number of cycles the instruction cache spends waiting for the ITLB to fill.
PM_EVENT_DTLBWAITCYCLES	26	Number of cycles the data cache spends waiting for the DTLB to fill.
PM_EVENT_UTLB ARBITRATIONCYCLES	27	Number of cycles where the ITLB or DTLB was waiting for access to the UTLB because the UTLB was busy servicing a request.
Reserved	28-31	Reserved for future use (on the ST230, counting reserved events has no effect, the counter does not increment).

All the events relating to the architectural state of the machine are sampled when bundles commit.

15.2 Access to registers

As all the performance monitoring registers are mapped into the control register space, access is only supported in supervisor mode. An attempt to read or write a register in user mode causes a CREG_ACCESS_VIOLATION exception.

15.3 Control register (PM_CR)

The program uses this control register to reset and enable all the counters, and define the events of the four programmable count registers. The control register's bit fields are listed in *Table 68*.

Table 68. PM_CR bit fields

Name	Bit(s)	Writable	Reset	Comment
ENB	0	RW	0x0	0: counting is disabled. 1: counting is enabled.
RST	1	RW	0x0	When a 1 is written all the counters (PM_CNT0-3 and PM_PCLK) are set to zero. If a 0 is written it is ignored. This field does not retain its value and so always reads as 0.
IDLE	2	RW	0x0	When the core enters idle mode, this bit is set to 1. Writing a 0 to this bit has no effect. Writing a 1 to this bit clears the bit.
Reserved	[11:3]	RO	0x0	Reserved
EVENT0	[16:12]	RW	0x0	5-bit field specifying the event being monitored for this counter.
EVENT1	[21:17]	RW	0x0	5-bit field specifying the event being monitored for this counter.
EVENT2	[26:22]	RW	0x0	5-bit field specifying the event being monitored for this counter.
EVENT3	[31:27]	RW	0x0	5-bit field specifying the event being monitored for this counter.

If counting is enable when the PM_CR register is written to any event that is triggered on the same bundle/cycle as the write is ignored. For example, if one of the event counters counts the load/store operations, a store to the PM_CR register is not included in the count.

Note: When the performance monitoring counters are enabled, the core does not enter idle mode, see Section 2.4.1: Idle mode macro on page 17.

15.4 Event counters (PM_CNT*i*)

Each of the four event counters is incremented by one each time the countable event specified in the PM_CR occurs. The four programmable event counters can record any one of the events specified in *Table 67 on page 112*.

Reading from these registers returns the current event count. Writing changes the current count. If a counter is written at the same time as an event triggers the counter to increment, then the increment is ignored.

If counting is enabled, when the counter is read the value of the counter does not include any event that was triggered on the same bundle/cycle as the read itself. For example, if an event counter was counting load/stores, the load that reads the count is not included in the count (but the event is still counted and will be available next time the counter is read).

15.5 Clock counter (PM_PCLK)

The PM_PCLK register is read/write. Reading the PM_PCLK register returns a 32-bit value. Writes to PM_PCLK update its value. This counter silently wraps back to zero when it overflows.

15.6 Recording events

To start recording, write the desired fields to an ST231 general purpose register. This can be achieved by first reading the PM_CR register, then modifying it as appropriate.

The ENB bit needs to be set to 1. The RST bit needs to be set to 1 if the counters are to be reset. The four programmable counter fields (EVENT*i* (where *i* = 0 to 3) of the PM_CR register) must be modified to the value representing the events to be counted. See the Value column in *Table 67: PM_EVENT values on page 112*.

The value in the register is then written to the memory mapped PM_CR for the operation to begin.

To stop recording, read the value of PM_CR, set the ENB bit to zero, and then write back to PM_CR. Do not change any other bits. If the RST bit is set to 1 then the PM_CNT*i* registers are reset.

Whilst counting events over a long period of time, the 32-bit counters may overflow. This overflow happens silently and the values wrap around to zero. To obtain a continuous profile, the counters must be read and reset at appropriate regular intervals (the exact interval depends upon the core clock frequency).

16 Execution model

This chapter defines how bundles are executed in terms of their component operations.

In the absence of traps, the core fetches a bundle from memory, decodes the operations within it and reads their operands. It then executes the operations in parallel and writes the results back to the architectural state of the machine. All operations in a bundle commit their results to the state of the machine at the same point in time. This is known as the commit point.

In the presence of traps, the core uses the commit point to distinguish between recoverable and non-recoverable traps.

Traps that are detected prior to the commit point are treated as recoverable. They are recoverable because the machine state has not been updated, which means that the state prior to the execution of the bundle can be recovered. In some cases, the cause of the trap can be corrected and the bundle restarted.

Traps detected after the commit point are unrecoverable. The machine state has been updated and in some cases it may not be clear which bundle caused the trap. Non-recoverable traps are consequently of a serious nature and cannot be restarted. On the ST231, the only class of non-recoverable trap is an error in the external memory system, which translates to a bus error exception.

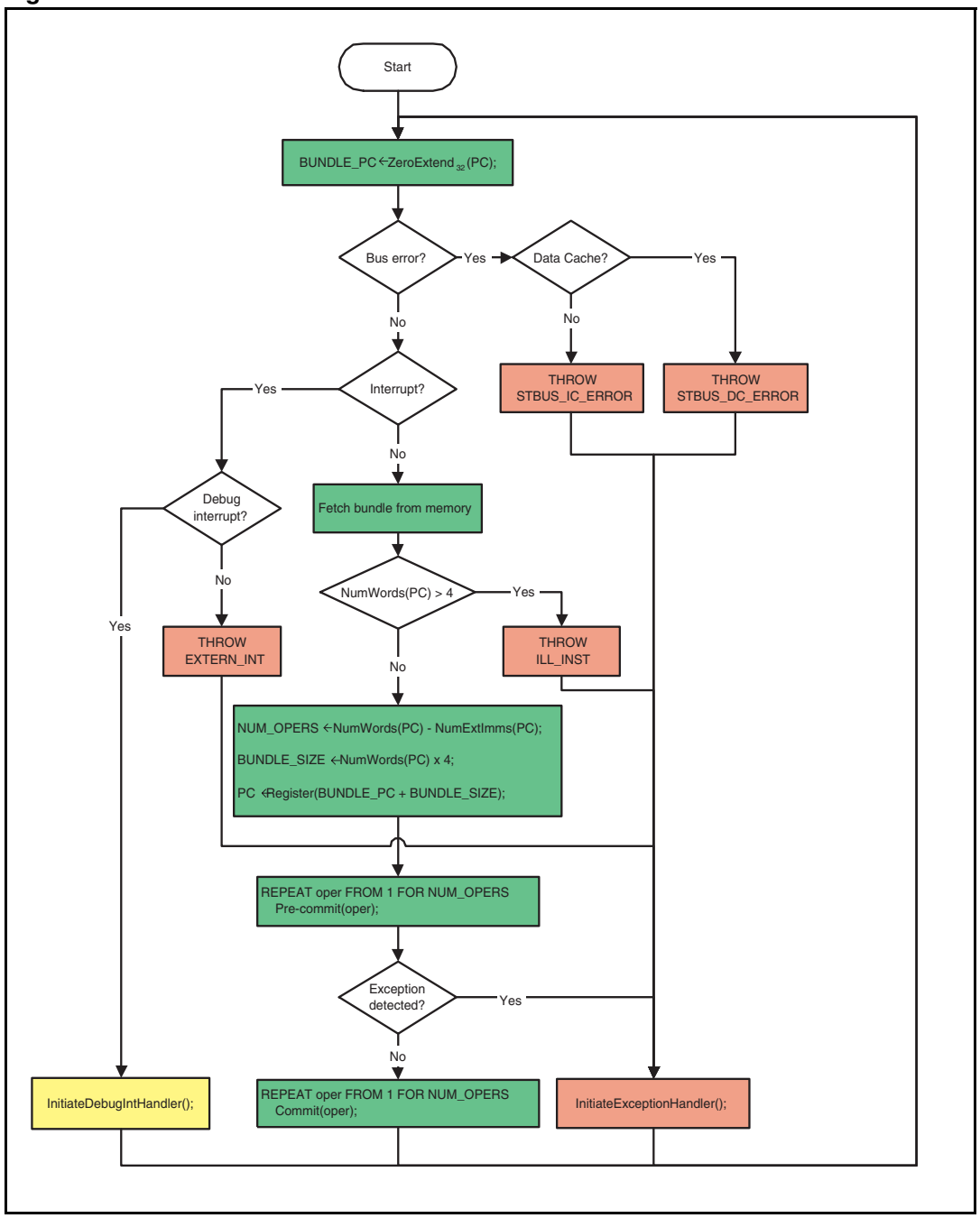
16.1 Bundle fetch, decode, and execute

The fetching, decoding and executing of bundles is specified using an abstract sequential model to show the effects on the architectural state of the machine. In this abstract model, each bundle is executed sequentially with respect to other bundles. This means that all actions associated with one bundle are completed before any actions associated with the next are started.

Specific implementations of the ST231 are generally designed to deliver substantial optimizations on the scheme provided by this abstract model. However, for legal bundle sequences that permit execution latency, these effects are not visible architecturally. The behavior of illegal cases is defined by *Chapter 5: Traps (exceptions and interrupts) on page 25*.

The execution flow shown in *Figure 18* uses notation defined in *Chapter 17: Specification notation on page 119*. There are additional functions that can be used to extract details from bundles. These are described in *Section 16.2: Functions on page 118*.

Figure 18. Execution model



16.2 Functions

The flow chart in *Figure 18* includes a number of functions that abstract out some the details. Those functions are described in this section. Starting with those used in the decode phase, then execution of operations, and finally the exceptional cases.

16.2.1 Bundle decode

The ST231 uses the functions listed in *Table 69* in the bundle decode phase.

Table 69. Bundle decode functions

Function	Description
NumWords(address)	Returns the number of words in the bundle. The return value is equal to the number of contiguous words, starting from **address**, without their **stop bit** set + 1.
NumExtImms(address)	Returns the number of extended immediates in the bundle starting at **address**.

16.2.2 Operation execution

The ST231 uses the functions listed in *Table 70* in the operation execution phase.

Table 70. Operation execution functions

Function	Description
Pre-commit(n)	For the operation **n**th operation in the bundle, execute the Pre-commit phase (*Section 18.2 on page 138*)(1).
Commit(n)	For the operation **n**th operation in the bundle, execute the Commit phase (*Section 18.2 on page 138*)(1).

1. Where **n** is in the range [1 ... number of operations in the bundle] inclusive.

16.2.3 Exceptional cases

The ST231 uses the functions listed in *Table 71* in exceptional cases.

Table 71. Operation execution functions

Function	Description
InitiateExceptionHandler()	Execute the statements defined in *Section 5.3: Saved execution state on page 26*.
InitiateDebugIntHandler()	Execute the statements defined in *Section 13.1.1: Debug interrupts on page 87*.

17 Specification notation

This chapter describes the formal language used in this manual for describing operations, exceptions and interrupts. The language has the following features:

- a simple variable and type system, see *Section 17.1*
- expressions, see *Section 17.2*
- statements, see *Section 17.3*
- notation for the architectural state of the machine, see *Section 17.4*

Additional mechanisms are defined to model memory (*Section 17.5.2*), control registers (*Section 17.5.3*), and cache instructions (*Section 17.5.4*).

Chapter 18: Instruction set on page 137 describes each instruction using informal text as well as the formal language. Occasionally it is not appropriate for one of these descriptions to describe the full semantics of the instruction; in such cases, both descriptions must be taken into account to constitute the full specification. In the case of an ambiguity or conflict, the notational language takes precedence over the text.

17.1 Variables and types

Variables are used to hold state. The type of a variable determines the set of values that the variable can take and the operators that can be applied to it. The scalar types are integers, booleans and bit-fields. One-dimensional arrays of scalar types are also supported.

The architectural state of the machine is represented by a set of variables. Each of these variables has an associated type, which is either a bit-field or an array of bit-fields. Bit-fields are used to give a bit-accurate representation of the variables.

The formal language uses additional variables to hold temporary values. The type of a temporary variable is determined by its context rather than explicit declaration. The type of a temporary variable is an integer, a boolean or an array of integers or boolean.

17.1.1 Integer

An integer variable can take the value of any mathematical integer. No limits are imposed on the range of integers supported. Integers obey their standard mathematical properties. Integer operations do not overflow. The integer operators are defined so that singularities do not occur. For example, no definition is given to the result of divide by zero; the operator is simply not available when the divisor is zero.

The representation of literal integer values is achieved using the following notations:

- Unsigned decimal numbers are represented by the regular expression: **[0-9]+**
- Signed decimal numbers are represented by the regular expression: **-[0-9]+**
- Hexadecimal numbers are represented by the regular expression: **0x[0-9a-fA-F]+**
- Binary numbers are represented by the regular expression: **0b[0-1]+**

These notations are standard and map onto integer values in the obvious way. Underscore characters (‘_’) can be inserted into any of the above literal representations. These do not change the represented value but can be used as spacers to aid readability.

17.1.2 Boolean

A boolean variable can take two values:

- boolean false: the literal representation of boolean false is **FALSE**
- boolean true: the literal representation of boolean true is **TRUE**

17.1.3 Bit-fields

Bit-fields are provided to define ‘bit-accurate’ storage.

Bit-fields containing arbitrary numbers of bits are supported. A bit-field of **b** bits contains bits numbered from **0** (the least significant bit) up to **b-1** (the most significant bit). Each bit can take the value **0** or the value **1**.

Bit-fields are mapped to, and from, unsigned integers in the usual way. If bit **i** of a **b**-bit bitfield, where **i** is in **[0, b)**, is set then it contributes 2^i to the integral value of the bitfield. The integral value of the bit-field as a whole is an integer in the range **[0, 2^b)**.

Bit-fields are mapped to, and from, signed integers using two’s complement representation. This is as above, except that the bit **b-1** of a **b**-bit bitfield contributes $-2^{(b-1)}$ to the integral value of the bitfield. The integral value of the bit-field as a whole is an integer in the range **[-2^{b-1}, 2^{b-1}]**.

A bitfield may be used in place of an integer value. In this case the integral value of the bitfield is used. A bit-field variable may be used in place of an integer variable as the target of an assignment. In this case the integer must be in the range of values supported by the bit-field.

17.1.4 Arrays

One-dimensional arrays of the above types are also available. Indexing into an **n**-element array **A** is achieved using the notation **A[i]** where **A** is an array of some type and **i** is an integer in the range **[0, n)**. This selects the $i^{th.}$ element of the array **A**. If **i** is zero this selects the first entry, and if **i** is **n-1** then this selects the last entry. The type of the selected element is the base type of the array.

Multi-dimensional arrays are not provided.

17.2 Expressions

Expressions are constructed from monadic operators, dyadic operators and functions applied to variables and literal values.

There are no defined precedence and associativity rules for the operators. Parentheses are used to specify the expression unambiguously.

Sub-expressions can be evaluated in any order. If a particular evaluation order is required, then sub-expressions must be split into separate statements.

17.2.1 Integer arithmetic operators

Since the notation uses straightforward mathematical integers, the set of standard mathematical operators is available and already defined.

The standard dyadic operators are listed in *Table 72*.

Table 72. Standard dyadic operators

Operation	Description
i + j	Integer addition
i - j	Integer subtraction
i × j	Integer multiplication
i / j	Integer division*
i \ j	Integer remainder*
* These operators are defined only for **j <> 0**	

The division operator truncates towards zero. The remainder operator is consistent with this. The sign of the result of the remainder operator follows the sign of the dividend. Division and remainder are not defined for a divisor of zero.

For a numerator (n) and a denominator (d), the following properties hold where d != 0:

$$n = d \times (n/d) + (n\backslash d)$$

$$(-n)/d = -(n/d) = n/(-d)$$

$$(-n)\backslash d = -(n\backslash d)$$

$$n\backslash(-d) = n\backslash d$$

$$0 <= (n\backslash d) < d \text{ where } n> = 0 \text{ and } d> 0$$

The standard monadic operators are described in *Table 73*.

Table 73. Standard monadic operators

Operator	Description
- i	Integer negation
|i|	Integer modulus (absolute value)

17.2.2 Integer shift operators

The available integer shift operators are listed in *Table 74*.

Table 74. Shift operators

Operation	Description
n << b	Integer left shift
n >> b	Integer right shift

The shift operators are defined on integers as follows where b >= 0:

$$n \ll b = n \times 2^{b}$$

$$n \gg b = \begin{cases} n / 2^{b} & \text{where} \quad n \geq 0 \\ (n - 2^{b} + 1) / 2^{b} & \text{where} \quad n < 0 \end{cases}$$

Right shifting by **b** places is a division by $\mathbf{2^b}$ but with the result rounded towards minus infinity. This contrasts with division, which rounds towards zero, and is the reason why the right shift definition is separate for positive and negative **n**.

17.2.3 Integer bitwise operators

The available integer bitwise operators are listed in *Table 75*.

Table 75. Bitwise operators

Operation	Description
i ∧ j	Integer bitwise **AND**
i ∨ j	Integer bitwise **OR**
i ⊕ j	Integer bitwise **XOR**
~ i	Integer bitwise **NOT**
$n_{<b\ FOR\ m>}$	Integer bit-field extraction: extract **m** bits starting at bit **b** from integer **n**
$n_{<b>}$	Integer bit-field extraction: extract **1** bit starting at bit **b** from integer **n**

In order to define bitwise operations all integers are considered as having an infinitely long two's complement representation. Bit 0 is the least significant bit of this representation, bit 1 is the next higher bit, and so on. The value of bit b, where b >= 0, in integer n is given by:

$$BIT(n, b) = (n/2^{b})\backslash 2 \qquad \text{where } n >= 0$$

$$BIT(n, b) = 1 - BIT((-n - 1), b) \qquad \text{where } n < 0$$

Care must be taken whenever the infinitely long two's complement representation of a negative number is constructed. This representation contains an infinite number of higher bits with the value **1** representing the sign. Typically, a subsequent conversion operation is used to discard these upper bits and return the result back to a finite value.

Bitwise **AND** (∧), **OR** (∨), **XOR** (⊕) and **NOT** (¬) are defined on integers as follows, where b takes all values such that b >= 0:

$$BIT(i \wedge j,\ b) = BIT(i,\ b) \times BIT(j,\ b)$$
$$BIT(i \vee\ j,\ b) = BIT(i \wedge j,\ b) + BIT(i \oplus j,\ b)$$
$$BIT(i \oplus j,\ b) = (BIT(i,\ b) + BIT(j,\ b))\backslash 2$$
$$BIT(\sim i,\ b) = 1 - BIT(i,\ b)$$

*Note: Bitwise **NOT** of any finite positive **i** results in a value containing an infinite number of higher bits with the value **1** representing the sign.*

Bitwise extraction is defined on integers as follows, where **b** ≥ **0** and **m** > **0**:

$$n\langle\ b\ \text{FOR}\ m\rangle = (n \gg b) \wedge (2^m - 1)$$
$$n\langle\ b\rangle = n\langle\ b\ \text{FOR}\ 1\rangle$$

The result of $\mathbf{n_{<b\ FOR\ m>}}$ is an integer in the range **[0, 2^m)**.

17.2.4 Relational operators

Relational operators are defined to compare integral values and give a boolean result. See *Table 76*.

Table 76. Relational operators

Operation	Description
i = j	Result is TRUE if i is equal to j, otherwise FALSE
i ! = j	Result is TRUE if i is not equal to j, otherwise FALSE
i < j	Result is TRUE if i is less than j, otherwise FALSE
i > j	Result is TRUE if i is greater than j, otherwise FALSE
i <= j	Result is TRUE if i is less than or equal to j, otherwise FALSE
i >= j	Result is TRUE if i is greater than or equal to j, otherwise FALSE

17.2.5 Boolean operators

Boolean operators are defined to perform logical **AND**, **OR**, **XOR** and **NOT**. These operators have boolean sources and result. Additionally, the conversion operator **INT** is defined to convert a boolean source into an integer result. See *Table 77*.

Table 77. Boolean operators

Operation	Description
i AND j	Result is TRUE if i and j are both true, otherwise FALSE
i OR j	Result is TRUE if either/both i and j are true, otherwise FALSE
i XOR j	Result is TRUE if exactly one of i and j are true, otherwise FALSE
NOT i	Result is TRUE if i is false, otherwise FALSE
INT i	Result is 0 if i is false, otherwise 1

17.2.6 Single-value functions

In some cases it is inconvenient or inappropriate to describe an expression directly in the specification language. In these cases a function call is used to reference the undescribed behavior.

A single-value function evaluates to a single value (the result), which can be used in an expression. The type of the result value can be determined by the expression context from which the function is called. There are also multiple-value functions which evaluate to multiple values. These are only available in an assignment context, and are described in *Section 17.3.2: Assignment on page 125*.

Functions may generate side-effects.

Arithmetic functions

Table 78. Arithmetic functions

Function	Description
`CountLeadingZeros(`*i*`)`	Convert integer *i* to 32-bit bitfield and return the number of leading zeros in the bitfield. For example: If $\mathbf{i_{<31>}}$ is **1** then the return value is **0.** If all bits are **0** then the return value is **32**.

Scalar conversions

Two monadic functions are defined to support conversion from integers to bit-limited signed and unsigned number ranges. For a bit-limited integer representation containing **n** bits, the signed number range is $\mathbf{[-2^{n-1}, 2^{n-1}]}$ while the unsigned number range is $\mathbf{[0, 2^{n}]}$.

These functions are often used to convert between signed and unsigned bit-limited integers and between bit-fields and integer values.

Table 79. Integer conversion operators

Function	Description
$\texttt{ZeroExtend}_n\texttt{(}i\texttt{)}$	Convert integer **i** to an **n**-bit 2's complement unsigned range
$\texttt{SignExtend}_n\texttt{(}i\texttt{)}$	Convert integer **i** to an **n**-bit 2's complement signed range

These two functions are defined as follows, where $\mathbf{n > 0}$:

$$\text{ZeroExtend}_n(i) = i_{\langle\ 0\ \text{FOR}\ n\rangle}$$

$$\text{SignExtend}_n(i) = \begin{cases} i_{\langle\ 0\ \text{FOR}\ n\rangle} & \text{where} \quad i_{\langle\ n-1\rangle} = 0 \\ i_{\langle\ 0\ \text{FOR}\ (n-1)\rangle} - 2^n & \text{where} \quad i_{\langle\ n-1\rangle} = 1 \end{cases}$$

For syntactic convenience, conversion functions are also defined for converting an integer or boolean to a single bit and to a value which can be stored as a 32-bit register. *Table 80* shows the additional functions provided.

Table 80. Conversion operators from integers to bit-fields

Operation	Description
`Bit(i)`	If *i* is a boolean, then this is equivalent to **`Bit(INT i)`** Otherwise, convert lowest bit of integer **i** to a 1-bit value This is a convenient notation for $\mathbf{i}_{<0>}$
`Register(i)`	If *i* is a boolean, then this is equivalent to `Register(INT i)` Otherwise, convert lowest 32 bits of integer **i** to an unsigned 32-bit value This is a convenient notation for $\mathbf{i}_{<0\ \mathbf{FOR}\ 32>}$

17.3 Statements

An instruction specification consists of a sequence of statements. These statements are processed sequentially in order to specify the effect of the instruction on the architectural state of the machine. The available statements are discussed in this section.

Each statement has a semi-colon terminator. A sequence of statements can be aggregated into a statement block using '**{**' to introduce the block and '**}**' to terminate the block. A statement block can be used anywhere that a statement can.

17.3.1 Undefined behavior

The statement:

```
UNDEFINED();
```

indicates that the resultant behavior is architecturally undefined.

A particular implementation can choose to specify an implementation-defined behavior in such cases. It is very likely that any implementation-defined behavior varies from implementation to implementation. Exploitation of implementation-defined behavior should be avoided to allow software to be portable between implementations.

In cases where architecturally undefined behavior can occur in user mode, the implementation ensures that implemented behavior does not break the protection model. Thus, the implemented behavior is some execution flow that is permitted for that user mode thread.

17.3.2 Assignment

The '←' operator is used to denote assignment of an expression to a variable. An example assignment statement is:

```
variable ← expression;
```

The expression can be constructed from variables, literals, operators and functions as described in *Section 17.2: Expressions on page 120*. The expression is fully evaluated before the assignment takes place. The variable can be an integer, a boolean, a bit-field or an array of one of these types.

Assignment to architectural state

This is where the variable is part of the architectural state, as described in *Table 81: Scalar architectural state on page 128*. The type of the expression and the type of the variable must match, or the type of the variable must be able to represent all possible values of the expression.

Assignment to a temporary

Alternatively, if the variable is not part of the architectural state, then it is a temporary variable. The type of the variable is determined by the type of expression. A temporary variable must be assigned to, before it is used in the instruction specification.

Assignment of an undefined value

An assignment of the following form results in a variable being initialized with an architecturally undefined value:

```
variable ← UNDEFINED;
```

After assignment the variable holds a value which is valid for its type. However, the value is architecturally undefined. The actual value can be unpredictable; that is to say the value indicated by UNDEFINED can vary with each use of UNDEFINED. Architecturally-undefined values can occur in both user and privileged modes.

A particular implementation can choose to specify an implementation-defined value in such cases. It is very likely that any implementation-defined values vary from implementation to implementation. If software is intended to be portable between ST231 implementation, then exploitation of implementation-defined values should be avoided.

Assignment of multiple values

Multi-value functions are used to return multiple values, and are only available when used in a multiple assignment context. The syntax consists of a list of comma-separated variables, an assignment symbol followed by a function call. The function is evaluated and returns multiple results into the variables listed. The number of variables and the number of results of the function must match. The assigned variables must all be distinct, that is, no aliases.

For example, a two-valued assignment from a function call with 3 parameters can be represented as:

```
variable1, variable2 ← call(param1, param2, param3);
```

17.3.3 Conditional

Conditional behavior is specified using `IF`, `ELSE IF` and `ELSE`.

Conditions are expressions that result in a boolean value. If the condition after an `IF` is true, then its block of statements is executed and the whole conditional is considered complete, ignoring any `ELSE IF` or `ELSE` clauses, if they exist. If the condition is false, then each of the `ELSE IF` clauses are processed, in turn, in the same manner. If no conditions are met and there is an `ELSE` clause then its block of statements is executed. Finally, if no conditions are met and there is no `ELSE` clause, then the statement has no effect apart from the evaluation of the condition expressions.

The ELSE IF and ELSE clauses are optional. In ambiguous cases, the ELSE matches with the preceding IF or ELSE IF.

For example:

```
IF (condition1)
  block1
ELSE IF (condition2)
  block2
ELSE
  block3
```

17.3.4 Repetition

Repetitive behavior is specified using the following construct:

```
REPEAT i FROM m FOR n STEP s
  block
```

The block of statements is iterated n times, with the integer i taking the values:

m, m +s, m + 2s, m + 3s, up to **m + (n - 1)** × **s**.

The behavior is equivalent to textually writing the block n times with i being substituted with the appropriate value in each copy of the block.

The value of n must be greater or equal to 0, and the value of s must be non-zero. The values of the expressions for m, n and s must be constant across the iteration. The integer **i** must not be assigned to within the iterated block. The STEP**s** can be omitted in which case the step-size takes the default value of **1**.

17.3.5 Exceptions

Exception handling is triggered by a THROW statement. When an exception is thrown, no further statements are executed from the operation specification; no architectural state is updated. Furthermore, if any one of the operations in a bundle triggers an exception, none of the operations update the architectural state.

If any operation in a bundle triggers an exception then an exception is taken. The actions associated with the taking of an exception are described in *Section 5.2: Exception handling on page 25*.

There are two forms of throw statement:

```
THROW type;
```

and:

```
THROW type, value;
```

where type indicates the type of exception which is launched, and value is an optional argument to the exception handling sequence. If **value** is not given, then it is undefined.

The exception types and priorities are described in detail in *Chapter 5: Traps (exceptions and interrupts) on page 25*.

17.3.6 Procedures

Procedure statements contain a procedure name followed by a list of comma-separated arguments contained within parentheses followed by a semi-colon. The execution of procedures typically causes side-effects to the architectural state of the machine.

Procedures are generally used where it is difficult or inappropriate to specify the effect of an instruction using the abstract execution model. A fuller description of the effect of the instruction is given in the surrounding text.

An example procedure with two parameters is:

```
proc(param1, param2);
```

17.4 Architectural state

Chapter 3: Architectural state on page 19 contains a full description of the visible state. The notations used in the specification to refer to this state are summarized in *Table 81* and *Table 82*. Each item of scalar architectural state is a bit-field of a particular width. Each item of array architectural state is an array of bit-fields of a particular width.

Table 81. Scalar architectural state

Architectural state	Type is a bit-field containing:	Description
PC	32 bits	Program counter; address of the current bundle
PSW	32 bits	Program status word
SAVED_PC	32 bits	Copy of the PC used during interrupts
SAVED_PSW	32 bits	Copy of the PSW used during interrupts
SAVED_SAVED_PC	32 bits	Copy of the PC used during debug interrupts
SAVED_SAVED_PSW	32 bits	Copy of the PSW used during debug interrupts
R*i* where *i* is in [0, 63]	32 bits	64 x 32-bit general purpose registers R0 reads as zero Assignments to R0 are ignored
LR	32 bits	Link register, synonym for R63
B*i* where *i* is in [0, 7]	1 bit	8 x 1-bit branch registers

Table 82. Array architectural state

Architectural state	Type is an array of bit-fields each containing:	Description
CR*i* where *i* is index of the control register	32 bits	Control registers, for which some specifications refer to individual control registers by their names as defined in the *Chapter 9: Control registers on page 67*.
MEM[i] where *i* is in [0, 2^{32}]	8 bits	2^{32} bytes of memory

17.5 Memory and control registers

This section describes the formal language defined to model memory (*Section 17.5.2 on page 130*), for control registers (*Section 17.5.3 on page 134*) and cache instructions (*Section 17.5.3 on page 134*).

17.5.1 Support functions

The functions used in the memory and control register descriptions are listed in *Table 83*.

Table 83. Support functions

Function	Description
`DataBreakPoint(`*address*`)`	Result is **TRUE** if `address` is in the range defined by data breakpoint control mechanism (*Section 13.1.2: Hardware breakpoint support on page 89*), otherwise **FALSE**
$\texttt{Misaligned}_{n}\texttt{(}$*address*$\texttt{)}$	Result is **TRUE** if `address` is not `n`-bit aligned, otherwise **FALSE**
`NoTranslation(`*address*`)`	Result is **TRUE** if the TLB is enabled and has no mapping for `address`, otherwise FALSE
`MultiMapping(`*address*`)`	Results is **TRUE** if the TLB has more than one mapping for `address`, otherwise **FALSE**
`Translate(`*address*`)`	Looks up `address` in the TLB and returns the associated physical address
`SCUHit(`*paddress*`)`	Results is **TRUE** if the physical address, `paddress`, hit the SCU, otherwise **FALSE**
`ReadAccessViolation(`*address*`)`	Result is **TRUE** if the TLB is enabled and a read access to **address** is not permitted by the TLB, otherwise **FALSE**
`WriteAccessViolation(`*address*`)`	Result is **TRUE** if the TLB is enabled and a write access to **address** is not permitted by the TLB, otherwise **FALSE**
`IsCRegSpace(`*address*`)`	Result is **TRUE** if `address` is in the control register space, otherwise **FALSE**
`UndefinedCReg(`*address*`)`	Result is **TRUE** if `address` does not correspond to a defined control register, otherwise **FALSE**
`CRegIndex(`*address*`)`	Returns the index of the control register which maps to `address`
`CRegReadAccessViolation(`*index*`)`	Result is **TRUE** if read access is not permitted to the given control register, otherwise **FALSE**
`CRegWriteAccessViolation(`*index*`)`	Result is **TRUE** if write access is not permitted to given control register, otherwise **FALSE**
`BusReadError(`*paddress*`)`	Result is **TRUE** if reading from physical address, `paddress`, generates a Bus Error, otherwise **FALSE**
`IsDBreakHit(`*address*`)`	Result is **TRUE** if `address` triggers a data breakpoint, otherwise it is **FALSE**

17.5.2 Memory model

The instruction specification uses a simple model of memory access which defines the relationship between the content of a logical memory and the values manipulated by instructions. The simple model ignores any caches that may be present; their operation is defined by the text of the architecture manual.

The processor's view of logical memory is defined in terms of an array **MEM[i]** defined in *Table 82: Array architectural state on page 128*. The mapping between the logical memory and a physical memory are described in *Appendix B: STBus endian behavior on page 320*.

The notation **MEM[s FOR n]** is used to denote an 8***n** bit bitfield produced from the concatenation of the **n** elements **MEM[s]** through **MEM[s+i-1]**, where **i** (the byte number) varies in the range **[0, n)**. The value of **MEM[s FOR n]** depends on the endianness of the processor.

- If the processor is operating in little endian mode then:

$$(\mathrm{MEM}[\mathrm{s\ FOR\ n}])\langle\ 8\mathrm{i\ FOR\ 8}\rangle = \mathrm{MEM}[\mathrm{s}+\mathrm{i}]$$

  This equivalence states that byte number **i** in the bit-field **MEM[s FOR n]** is the **i**$^{th.}$ byte in memory counting upwards from **MEM[s]**.

- If the processor is operating in big endian mode then:

$$(\mathrm{MEM}[\mathrm{s\ FOR\ n}])\langle\ 8\mathrm{i\ FOR\ 8}\rangle = \mathrm{MEM}[(\mathrm{s}+\mathrm{n}-1)-\mathrm{i}]$$

  This equivalence states that byte number **i**, using big endian byte numbering (that is, byte **0** is bits **8n-8** to **8n-1**), in the bit-field **MEM[s FOR n]** is the **i**$^{th.}$ byte in memory counting downwards from **MEM[n]**.

For syntactic convenience, functions and procedures are provided to read and write memory.

Support functions

The specification of the memory instructions relies on the support functions listed in *Table 83: Support functions on page 129*. These functions are used to model the behavior of the TLB described in *Chapter 6: Memory translation and protection on page 31*.

Reading memory

The functions provided to support the reading of memory are listed in *Table 84*.

Table 84. Memory read functions

Function	Description
`ReadCheckMemory`$_n$`(`*address*`)`	Throws any non-BusError exception generated by an `n`-bit read from `address`.
`PrefetchCheckMemory(`*address*`)`	Throws any BusError exceptions generated by a prefetch from `address`.
`ReadMemory`$_n$`(`*address*`)`	Issues an `n`-bit read to `address` (can generate BusError exception).
`DisReadCheckMemory`$_n$`(`*address*`)`	Throws any non-BusError exception generated by an **n**-bit dismissible read from `address`.
`DisReadMemory`$_n$`(`*address*`)`	Returns either `n`-bits from `address` or `0` (can generate BusError exception).
`ReadMemResponse()`	Returns the value of the read request issued.

The `ReadCheckMemory`$_n$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being read (`n`) is one of 8, 16, or 32. The procedure throws any alignment or access violation exceptions generated by a read access to that address.

```
ReadCheckMemory_n(a);
```

is equivalent to:

```
IF (Misaligned_n(a))
  THROW MISALIGNED_TRAP, a;
IF (PSW[TLB_ENABLE])
  IF (NoTranslation(a) OR
    MultiMapping(a) OR
    ReadAccessViolation(a))
    THROW DTLB, a;
```

Similarly, if the memory access is a dismissible read:

```
DisReadCheckMemory_n(a);
```

is equivalent to:

```
IF (Misaligned_n(a) AND PSW[SPECLOAD_MALIGNTRAP_EN])
  THROW MISALIGNED_TRAP, a;
IF (PSW[TLB_ENABLE]) {
  IF (MultiMapping(a))
    THROW DTLB, a;
  IF (PSW[TLB_DYNAMIC] AND NoTranslation(a))
       THROW DTLB, a;
}
```

The `ReadMemory`$_{\mathbf{n}}$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being read (**n**) is one of **8**, **16**, or **32**. The required bytes are read from memory, interpreted according to endianness, and the read bit-field value assigned to a temporary integer. If the read memory value is to be interpreted as signed, then a sign-extension should be used when accessing the result using `ReadMemResponse`. The procedure call:

```
ReadMemory_n(a);
```

is equivalent to:

```
pa = Translate(a);
width ←n / 8;
IF (BusReadError(pa))
   THROW BUS_DC_ERROR, a; // Non-recoverable
mem_response ←MEM[pa FOR width];
```

The `DisReadMemory`$_{\mathbf{n}}$ performs the same functionality for a dismissible read from memory. The procedure call:

```
DisReadMemory_n(a);
```

is equivalent to:

```
width ←n / 8;
IF (NOT Misaligned_n(a) AND
   NOT NoTranslation(a) AND
   NOT ReadAccessViolation(a) {
      pa = Translate(a);
      IF (SCUHit(pa) {
         IF (BusReadError(pa))
            THROW BUS_DC_ERROR, a; // Non-recoverable
         mem_response ←MEM[pa FOR width];
      }
      ELSE
         mem_response ←0;
   }
ELSE
   mem_response ←0;
```

The function `ReadMemResponse` returns the data that has been read from memory. The assignment:

```
result ←ReadMemResponse();
```

is equivalent to:

```
result ←mem_response;
```

Prefetching memory

Table 85 describes the procedure that is provided to denote memory prefetch.

Table 85. Memory prefetch procedure

Function	Description
`PrefetchMemory(address)`	Prefetch memory if possible.

This is used for a software-directed data prefetch from a specified effective address. This is a hint to give advance notice that particular data will be required. `PrefetchMemory`, performs the implementation-specific prefetch when the address is valid:

```
PrefetchMemory(a);
```

This is equivalent to:

```
IF (NOT NoTranslation(a) AND
  NOT MultiMapping(a)
  NOT ReadAccessViolation(a)) {
    pa = Translate(a);
    IF (SCUHit(pa))
      Prefetch(a);
  }
```

where `Prefetch` is a cache operation defined in *Section 17.5.4: Cache model on page 136*. Prefetching memory does not generate any exceptions.

Writing memory

Table 86 lists the procedures that are provided to write memory.

Table 86. Memory write procedures

Function	Description
$\texttt{WriteCheckMemory}_{n}\texttt{(}\textit{address}\texttt{)}$	Throws any exception generated by an *n*-bit write to *address*
$\texttt{WriteMemory}_{n}\texttt{(}\textit{address, value}\texttt{)}$	Aligned *n*-bit write to memory

The $\texttt{WriteCheckMemory}_{\mathbf{n}}$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being written (n) is one of **8**, **16**, or **32**. The procedure throws any alignment or access violation exceptions generated by a write access to that address.

$\texttt{WriteCheckMemory}_{n}\texttt{(a);}$

This is equivalent to:

```
IF (Misaligned_n(a))
  THROW MISALIGNED_TRAP, a;
IF (NoTranslation(a) OR
  MultiMapping(a) OR
  WriteAccessViolation(a))
  THROW DTLB, a;
```

The `WriteMemory`$_{\mathbf{n}}$ procedure takes an integer parameter to indicate the address being accessed, followed by an integer parameter containing the value to be written. The number of bits being written (**n**) is one of **8**, **16**, **32** or **64** bits. The written value is interpreted as a bit-field of the required size; all higher bits of the value are discarded. The bytes are written to memory, ordered according to endianness. The statement:

```
WriteMemory_n(a, value);
```

This is equivalent to:

```
pa = Translate(a);
width ←n / 8;
MEM[pa FOR width] ←value<0 FOR n>;
```

17.5.3 Control register model

This section describes the control register model; how control registers are read and written to.

Reading control registers

The procedures listed in *Table 87* are provided to read from control registers.

Note: Only word (32-bit) control register accesses are supported.

Table 87. Control register read functions

Function	Description
`ReadCheckCReg(`*`address`*`)`	Throws any exception generated by reading from *address* in the control register space
`ReadCReg(`*`address`*`)`	Issues a read from the control register mapped to *address*

The `ReadCheckCReg` procedure takes an integer parameter to indicate the address being accessed. The procedure throws any alignment or non-mapping exception generated by reading from the control register space.

```
ReadCheckCReg(a);
```

This is equivalent to:

```
IF (UndefinedCReg(a))
  THROW CREG_NO_MAPPING, a;
index ←CRegIndex(a);
IF (CRegReadAccessViolation(index))
  THROW CREG_ACCESS_VIOLATION, a;
```

The control register file is denoted **CR**. The function `ReadCReg` is provided:

```
ReadCReg(a);
```

This is equivalent to:

```
index ←CRegIndex(a);
mem_response ←CR_index;
```

Writing control registers

The procedures listed in *Table 88* are provided to read from control registers. Note that only word (32-bit) control register accesses are supported.

Table 88. Control registers write procedures

Function	Description
`WriteCheckCReg(address)`	Throws any exception generated by writing to the *address* in the control register space
`WriteCReg(address, value)`	Writes *value* to the control register mapped to *address*

The WRITECHECKCREG procedure takes an integer parameter to indicate the address being accessed. The procedure throws any alignment, non-mapping or access violation exceptions generated by writing to the control register space:

```
WriteCheckCReg(a);
```

is equivalent to:

```
IF (UndefinedCReg(a))
  THROW CREG_NO_MAPPING, a;
index ←CRegIndex(a);
IF (CRegWriteAccessViolation(index))
  THROW CREG_ACCESS_VIOLATION, a;
```

A procedure called WRITECREG is provided to write control registers:

```
WriteCReg(a, value);
```

is equivalent to:

```
index ←CRegIndex(a);
CR_index ←value;
```

17.5.4 Cache model

The core uses cache operations to prefetch and purge lines in caches. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the *Chapter 7: Memory subsystem on page 48*.

Table 89. Procedures to model cache operations

Procedure	Description
`PurgeIns()`	Invalidate the entire instruction cache (see *Invalidating the entire instruction cache on page 50*).
`Sync()`	Data memory subsystem synchronization function (see *Section 7.3.8: D-side synchronization on page 54*).
`PurgeAddressCheckMemory(`*`address`*`)`	Throws any exceptions generated by purging addresses from the data cache (see *Purging data by address on page 54*).
`PurgeAddress(`*`address`*`)`	Purge address from the data cache (see *Purging data by address on page 54*).
`PurgeSet(`*`address`*`)`	Purge a set of lines from the data cache (see *Purging data by set on page 54*).
`Prefetch(`*`address`*`)`	Prefetch a data cache line if it is in cacheable memory (see *Section 7.3.6: Prefetching data on page 53*).
`PurgeInsPg(`*`address`*`)`	Purges the given virtual/physical address combination 8 Kbyte page from the instruction cache (see *Invalidating the instruction cache by page on page 50*).

17.5.5 Architectural state model

Architectural state such as the PC and PSW is modified by a number of procedures. These also have the effect of flushing the pipeline; this is beyond the scope of the specification language.

Table 90. Procedures to model changing architectural state

Procedure	Description
`Rfi()`	Return from interrupt. This flushes the pipeline (see *Section 5.3: Saved execution state on page 26*).
`PswSet(`*`value`*`)`	`PSW <- PSW |` *`value`*. This flushes the pipeline (see *Section 3.4.4: PSW access on page 21*). If value is zero then this flushes out any unexecuted syllables so that the next bundle is guranteed to be fetched from the instruction cache.
`PswClr(`*`value`*`)`	`PSW <- PSW & (~`*`value`*`)`. This flushes the pipeline (see *Section 3.4.4: PSW access on page 21*).

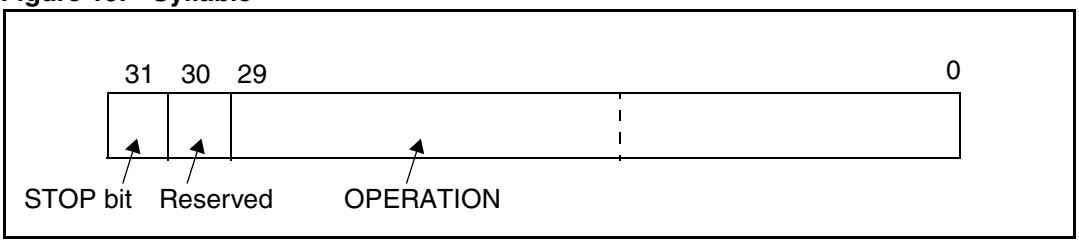
18 Instruction set

This chapter contains descriptions of all the operations and macros (pseudo-operations) in the ST231 instruction set. *Section 18.1: Bundle encoding* has been included in order to describe how operations are encoded in the context of bundles.

18.1 Bundle encoding

An instruction bundle consists of between one and four consecutive 32-bit words, known as syllables. Each syllable encodes either an operation or an extended immediate. The most significant bit of each syllable (bit 31) is a **stop bit** which is set to indicate that it is the last in the bundle, as shown in *Figure 19*.

Figure 19. Syllable



18.1.1 Extended immediates

Many operations have an **Immediate** form. In general only small (9-bit) immediates can be directly encoded in a single word syllable. In the event that larger immediates are required, an immediate extension is used. This extension is encoded in an adjacent word in the bundle, making the operation effectively a two-word operation.

These immediate extensions associate either with the operation to their left or their right in the bundle. Bit 23 is used to indicate the association.

- 0 indicates left association (word address - 1) (**imml**)
- 1 indicates right association (word address + 1) (**immr**)

The semantic descriptions of **Immediate** form operations use the following function to take into account possible immediate extensions, as shown in *Table 91*.

Table 91. Extended immediate functions

Function	Description
`Imm(`*`i`*`)`	Given short immediate value *i*, returns an integer value that represents the full immediate.

This function effectively performs the following:

If there is an **immr** word to the left (word address - 1) or an **imml** word to the right (word address + 1) in the bundle, then **Imm** returns:

```
(ZeroExtend23(extension) << 9) + ZeroExtend9(i);
```

Where `extension` represents the lower 23 bits of the associated **immr** or **imml**.

Otherwise **Imm** returns:

```
SignExtend9(i);
```

18.1.2 Encoding restrictions

There are a number of restrictions placed on the encoding of bundles. It is the duty of the assembler to ensure that these restriction are obeyed.

- Long immediates must be encoded at even word addresses.
- Multiply operations must be encoded at odd word addresses.
- There may only be one control flow operation per bundle, and it must be the first syllable.
- There may only be one **load** or **store** operation per bundle.

18.2 Operation specifications

The specification of each operation contains the following fields:

- Name: the name of the operation with an optional subscript. The subscript distinguishes between operations with different operand types. For example, integer operations can have either **Register** or **Immediate** formats. If no subscript exists for an operation, then there is only one format.
- Syntax: presents the assembly syntax of the operation (*ST200 Programming Manual*)
- Encoding: the binary encoding is summarized in a table. It shows which bits are used for the opcode, which bits are reserved (empty fields) and which bit-fields encode the operands. The operands are either register designators or immediate constants.
- Semantics: a table containing the statements (*Section 17.3: Statements on page 125*) that define the operation. The notation used is defined in *Chapter 17: Specification notation on page 119*. The table is divided into two parts by the commit point. (See *Chapter 16: Execution model on page 116*.)

Pre-commit phase: ● No architectural state of the machine is updated. ● Any recoverable exceptions are thrown here.	
Commit phase - executed if no exceptions have been thrown: ● All architectural state is updated. ● Any exceptions thrown here are non-recoverable(1).	←Commit point

1. For the ST231 the only non-recoverable exception is a bus error.

- Description: a brief textual description of the operation

- Restrictions: contains any details of restrictions, these may be of the following types:
  - Address/bundle: In encoding a bundle with the operation there are a number of possible restrictions which may apply, see *Section 18.1.2*
  - Latency: certain operands have latency constraints that must be observed
  - Destination restrictions: certain operations are not allowed to use the Link Register (LR) as a destination
- Exceptions: if this operation is able to throw any exceptions, they are listed here. The semantics of the operation detail how and when they are thrown

18.3 Example operations

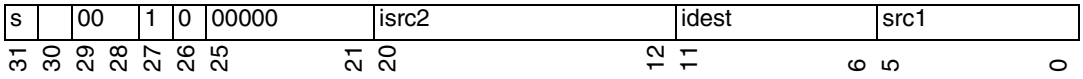
18.3.1 add Immediate

The specification for this operation is shown in *Figure 20*.

Figure 20. Example operation

add Immediate

add $R_{IDEST} = R_{SRC1}, I_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 + operand2;

R_{IDEST} ←Register(result1);

Description: Add

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

The operation is given the subscript **Immediate** to indicate that one of its source operands is an immediate rather than both being registers.

The next line of the description shows the assembly syntax of the operation.

Just below is the binary encoding table with fields showing:

- The opcode: Bits 29:21
- The operands: An **s** in bit 31 represents the **stop bit** (*Section 18.1*)
  - The 9-bit immediate constant, bits 20:1
  - The destination register designator, bits 11:6
  - The source register designator, bits 5:0
- Unused bits: Bit 30

The semantics table specifies the effects of the executing this operation. The table is divided into two parts. The first half containing statements which do not affect the architectural state of the machine. The second half containing statements that will not be executed if an exception occurs in the bundle.

The statements themselves are organized into 3 stages as follows:

1. The first two statements read the required source information:

   ```
   operand1 <- SignExtend32(RSRC1);
   operand2 <- Imm(ISRC2);
   ```

   The first statement reads the value of the R_{SRC1} register, interprets it as a signed 32-bit value and assigns this to a temporary integer called **operand1**. The second statement passes the value of ***ISRC2*** to the immediate handling function **Imm** (*Section 18.1.1*). The result of the function is interpreted as a signed 32-bit value and assigned to a temporary integer called **operand2**.

2. The next statement implements the addition:

   ```
   result <- operand1 + operand2;
   ```

   This statement does not refer to any architectural state. It adds the 2 integers **operand1** and **operand2** together, and assigns the result to a temporary integer called **result**. Note that since this is a conventional mathematical addition, the result can contain more significant bits of information than the sources.

3. The final statement, executed if no exceptions have been thrown in the bundle, updates the architectural state:

   ```
   RIDEST <- Register(result);
   ```

   The function **Register** (*Section 17.2.6: Single-value functions on page 124*) converts the integer **result** back to a bit-field, discarding any redundant higher bits. This value is then assigned to the R_{IDEST} register.

After the semantic description is a simple textual description of the operation.

The restrictions section shows that this operation has no restrictions. This means that up to four of these operations can be used in a bundle, and that all operands are ready for use by operations in the next bundle.

Finally, this operation can not generate any exceptions.

18.4 Macros

Table 92 is a list of the currently implemented pseudo-operations or ‘macros’. Each macro is essentially a simplified synonym for another, less intuitive operation.

Table 92. Macros

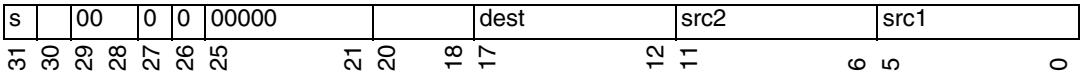
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
convbi	s		01		1	001			scond			000000001									idest						000000				
	slctf R_{IDEST} = B_{SCOND}, R_0, 1																															
convib	s		00		0	1	1	1100				000000					bdest				000000						src1				
	orl B_{BDEST2} = R_{SRC1}, R_0																															
idle	1		11		0	001			0	00000000000000000000																						
	goto 0																															
mfb	s		01		1	001			scond			000000001									idest						000000				
	slctf R_{IDEST} = B_{SCOND}, R_0, 1																															
mov_{bsrc}	s		01		1	001			scond			000000001									idest						000000				
	slctf R_{IDEST} = B_{SCOND}, R_0, 1																															
mov_{bdest}	s		00		0	1	1	1100				bdest									000000						src1				
	orl B_{BDEST} = R_{SRC1}, R_0																															
mov_{Int3R}	s		00		0	0		00000									dest				src2						000000				
	add R_{DEST} = R_0, R_{SRC2}																															
mov_{Int3I}	s		00		1	0		00000									isrc2				idest						000000				
	add R_{IDEST} = R_0, ISRC2																															
mtb	s		00		0	1	1	1100				bdest									000000						src1				
	orl B_{BDEST} = R_{SRC1}, R_0																															
nop	s		00		0	0		00000									000000				000000						000000				
	add R_0 = R_0, R_0																															
return	s		11		0	001			1	btarg																						
	goto $r63																															
syncins	s		10		10010																000000						src1				
	pswset R_0																															
zxtb	s		00		1	0		01001				011111111									idest						src1				
	and R_{IDEST} = R_{SRC1}, 255																															

18.5 Operations

Each operation is now specified. They are listed alphabetically for ease of use. The semantics of the operations are written using the notational language defined in *Chapter 17: Specification notation on page 119*.

add Register

add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 + operand2;
R_{DEST} ←Register(result1);

Description: Add

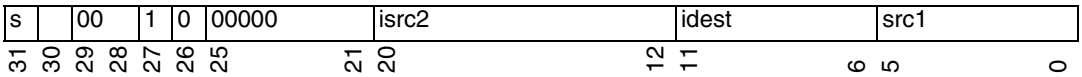
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

add Immediate

add $R_{IDEST} = R_{SRC1}, I_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 + operand2;
R_{IDEST} ←Register(result1);

Description: Add

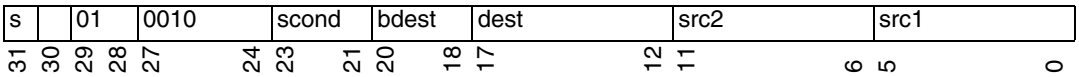
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

addcg

addcg R_{DEST}, B_{BDEST} = R_{SRC1}, R_{SRC2}, B_{SCOND}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); operand3 ←ZeroExtend$_{1}$(B_{SCOND}); result1 ←(operand1 + operand2) + operand3; result2 ←Bit(result1, 32);
R_{DEST} ←Register(result1); B_{BDEST} ←Bit(result2);

Description: Add with carry and generate carry

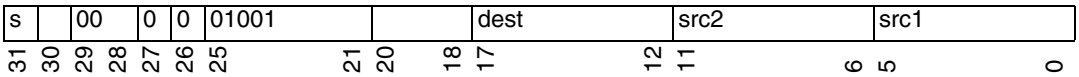
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

and Register

and $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ∧ operand2;

R_{DEST} ←Register(result1);

Description: Bitwise and

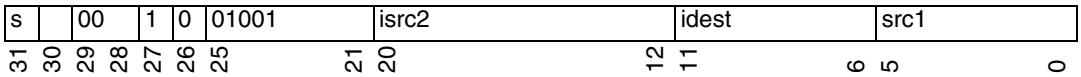
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

and Immediate

and R_{IDEST} = $R_{SRC1,}$ ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 ∧ operand2;
R_{IDEST} ←Register(result1);

Description: Bitwise and

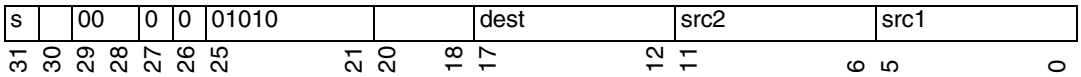
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

andc Register

andc $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(~operand1) ∧ operand2;

R_{DEST} ←Register(result1);

Description: Complement and bitwise and

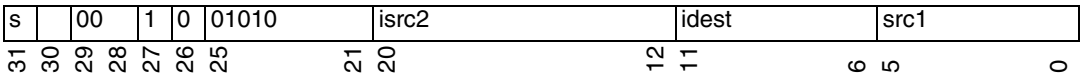
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

andc Immediate

andc R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←(~operand1) ∧ operand2;
R_{IDEST} ←Register(result1);

Description: Complement and bitwise and

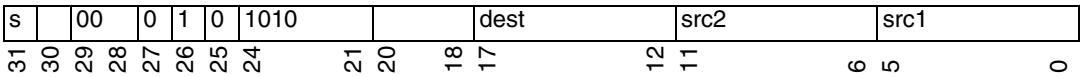
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

andl Register - Register

andl $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ! = 0) AND (operand2 ! = 0);

R_{DEST} ←Register(result1);

Description: Logical and

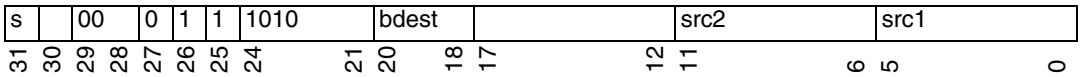
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

andl Branch Register - Register

andl $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ! = 0) AND (operand2 ! = 0);

B_{BDEST} ←Bit(result1);

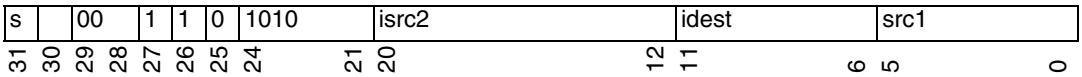
Description: Logical and

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

andl Register - Immediate
andl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 ! = 0) AND (operand2 ! = 0);

R_{IDEST} ←Register(result1);

Description: Logical and

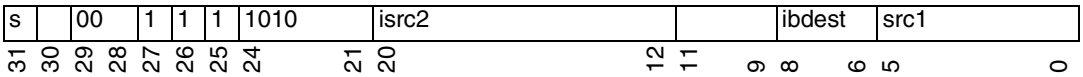
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

andl Branch Register - Immediate

andl $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←$SignExtend_{32}(R_{SRC1})$;
operand2 ←$SignExtend_{32}$(Imm(ISRC2));
result1 ←(operand1 ! = 0) AND (operand2 ! = 0);

B_{IBDEST} ←Bit(result1);

Description: Logical and

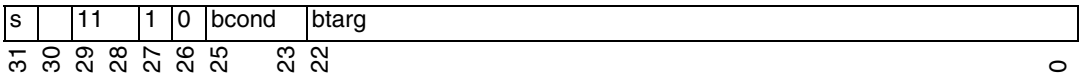
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

br

br B_{BCOND}, BTARG



Semantics:

operand1 ←ZeroExtend$_1$(B_{BCOND});
operand2 ←SignExtend$_{23}$(BTARG)<< 2;
IF (operand1 ! = 0)
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + operand2);

Description: Branch

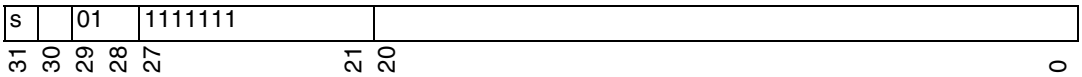
Restrictions: Must be the first syllable of a bundle.

There is a latency of 2 bundles between an operation writing B_{BCOND} and this operation being issued.

Exceptions: None.

break

break



Semantics:

```
THROW ILL_INST;
```

Description: Break

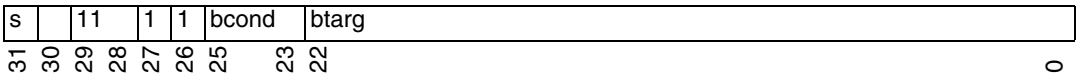
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: ILL_INST

brf

brf B_{BCOND}, BTARG



Semantics:

operand1 ←ZeroExtend$_1$(B_{BCOND});
operand2 ←SignExtend$_{23}$(BTARG)<< 2;
IF (operand1 = 0)
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + operand2);

Description: Branch false

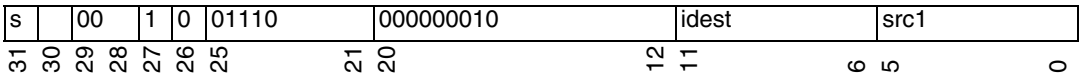
Restrictions: Must be the first syllable of a bundle.

There is a latency of 2 bundles between an operation writing B_{BCOND} and this operation being issued.

Exceptions: None.

bswap

bswap $R_{IDEST} = R_{SRC1}$



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
byte0 ←operand1<0 FOR 8>;
byte1 ←operand1<8 FOR 8>;
byte2 ←operand1<16 FOR 8>;
byte3 ←operand1<24 FOR 8>;
result1 ←((byte0 << 24) ∨ (byte1 << 16)) ∨ ((byte2 << 8) ∨ byte3);
```

```
RIDEST ←Register(result1);
```

Description: Byte swap

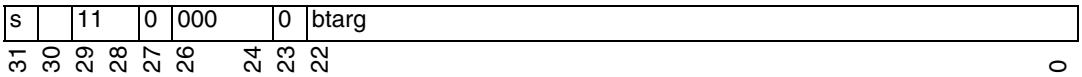
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

call Immediate

call $r63 = BTARG



Semantics:

operand1 ←SignExtend$_{23}$(BTARG)<< 2;
NEXT_PC←PC;
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + operand1);
LR ←NEXT_PC;

Description: Jump and link

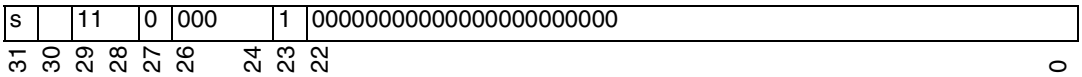
Restrictions: Must be the first syllable of a bundle.

No latency constraints.

Exceptions: None.

call Link Register

call $r63 = $r63



Semantics:

```
NEXT_PC←PC;
PC ←Register(ZeroExtend32(LR));
LR ←NEXT_PC;
```

Description: Jump (using Link Register) and link

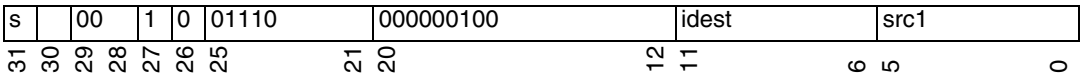
Restrictions: Must be the first syllable of a bundle.

There are no latency constraints between an call uprating the LR and this operation. There is a latency of 4 bundles between a load writing to the LR and this operation. There is a latency of 2 bundles between any other operation updating the LR and this operation.

Exceptions: None.

clz

clz R_{IDEST} = R_{SRC1}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); result1 ←CountLeadingZeros(operand1);
R_{IDEST} ←Register(result1);

Description: Count leading zeros

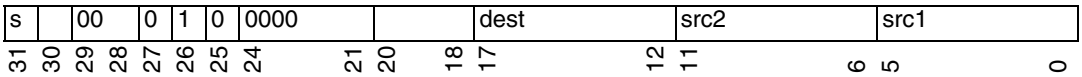
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpeq Register - Register

cmpeq $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 = operand2;
R_{DEST} ←Register(result1);

Description: Test for equality

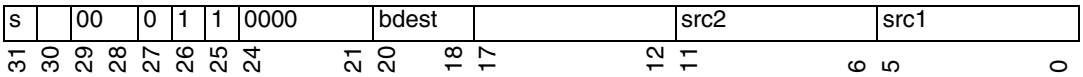
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpeq Branch Register - Register

cmpeq $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 = operand2;
B_{BDEST} ←Bit(result1);

Description: Test for equality

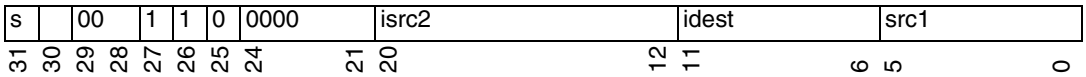
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpeq Register - Immediate

cmpeq R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 = operand2;

R_{IDEST} ←Register(result1);

Description: Test for equality

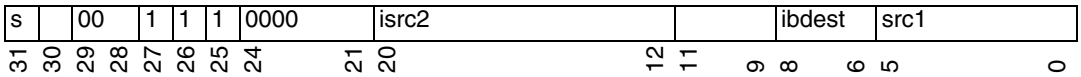
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpeq Branch Register - Immediate

cmpeq $B_{IBDEST} = R_{SRC1}, I_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 = operand2;
B_{IBDEST} ←Bit(result1);

Description: Test for equality

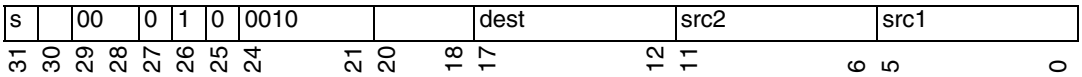
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpge Register - Register

cmpge $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 ≥ operand2;
R_{DEST} ←Register(result1);

Description: Signed compare equal or greater than

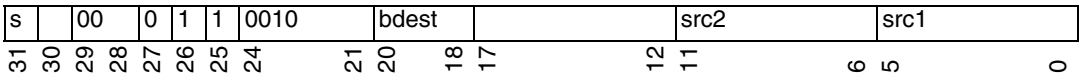
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpge Branch Register - Register

cmpge $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 ≥ operand2;
B_{BDEST} ←Bit(result1);

Description: Signed compare equal or greater than

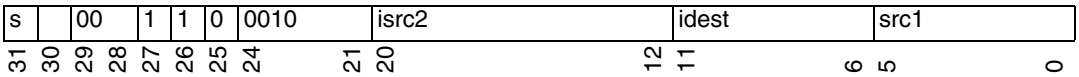
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpge Register - Immediate

cmpge R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 $\succeq$ operand2;
R_{IDEST} ←Register(result1);

Description: Signed compare equal or greater than

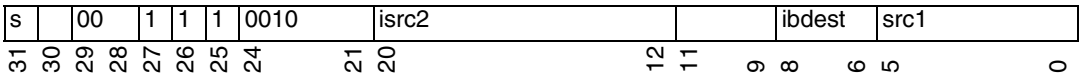
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpge Branch Register - Immediate

cmpge B_{IBDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 ≥= operand2;
B_{IBDEST} ←Bit(result1);

Description: Signed compare equal or greater than

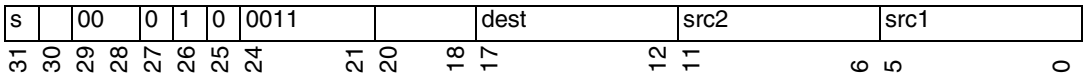
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgeu Register - Register

cmpgeu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result1 ←operand1 ≥= operand2;
R_{DEST} ←Register(result1);

Description: Unsigned compare equal or greater than

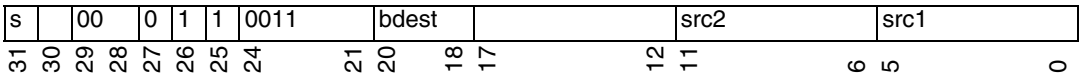
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgeu Branch Register - Register

cmpgeu $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←operand1 $\succeq$ operand2;

B_{BDEST} ←Bit(result1);

Description: Unsigned compare equal or greater than

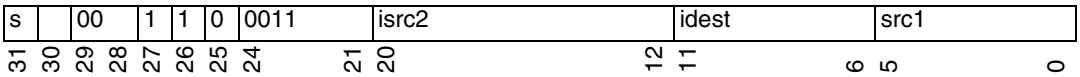
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgeu Register - Immediate

cmpgeu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≍ operand2;

R_{IDEST} ←Register(result1);

Description: Unsigned compare equal or greater than

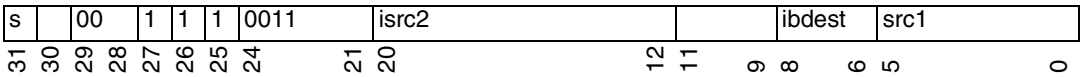
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgeu Branch Register - Immediate

cmpgeu $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 $\succeq$ operand2;

B_{IBDEST} ←Bit(result1);

Description: Unsigned compare equal or greater than

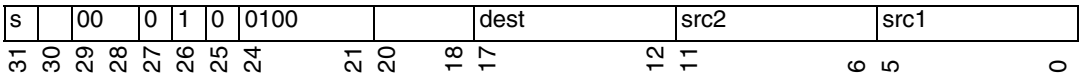
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgt Register - Register

cmpgt R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 > operand2;
R_{DEST} ←Register(result1);

Description: Signed compare greater than

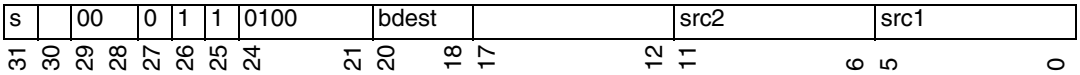
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgt Branch Register - Register

cmpgt B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 > operand2;
B_{BDEST} ←Bit(result1);

Description: Signed compare greater than

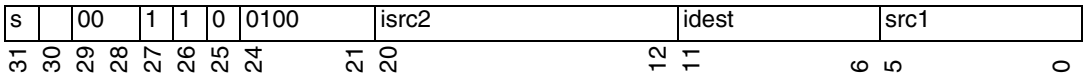
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgt Register - Immediate

cmpgt $R_{IDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 > operand2;
R_{IDEST} ←Register(result1);

Description: Signed compare greater than

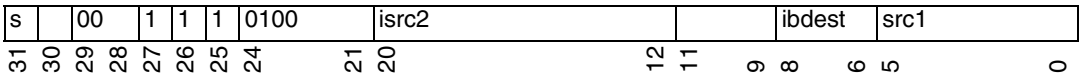
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgt Branch Register - Immediate

cmpgt $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 > operand2;
B_{IBDEST} ←Bit(result1);

Description: Signed compare greater than

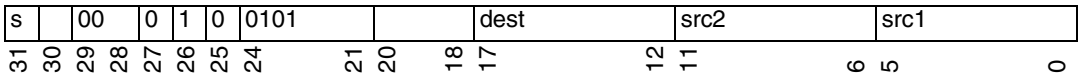
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgtu Register - Register

cmpgtu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result1 ←operand1 > operand2;
R_{DEST} ←Register(result1);

Description: Unsigned compare greater than

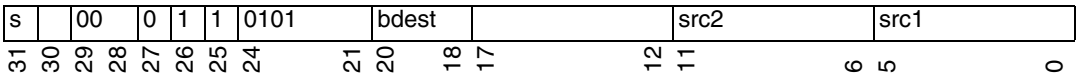
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgtu Branch Register - Register

cmpgtu $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←operand1 > operand2;

B_{BDEST} ←Bit(result1);

Description: Unsigned compare greater than

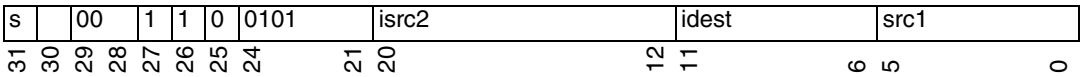
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgtu Register - Immediate

cmpgtu $R_{IDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 > operand2;
R_{IDEST} ←Register(result1);

Description: Unsigned compare greater than

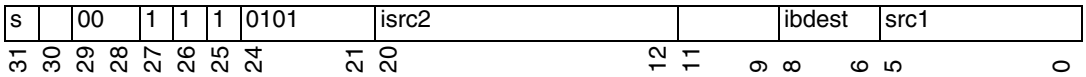
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpgtu Branch Register - Immediate

cmpgtu $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 > operand2;
B_{IBDEST} ←Bit(result1);

Description: Unsigned compare greater than

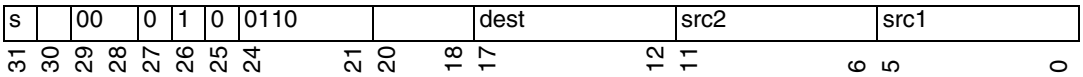
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmple Register - Register

cmple $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 <= operand2;
R_{DEST} ←Register(result1);

Description: Signed compare equal or less than

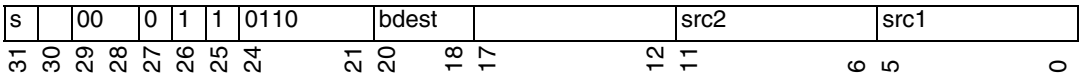
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmple Branch Register - Register

cmple $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 <= operand2;
B_{BDEST} ←Bit(result1);

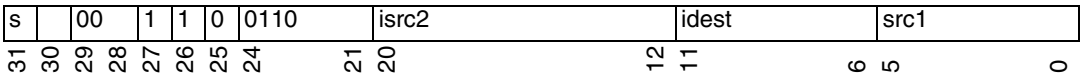
Description: Signed compare equal or less than

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmple Register - Immediate
cmple R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 <= operand2;
R_{IDEST} ←Register(result1);

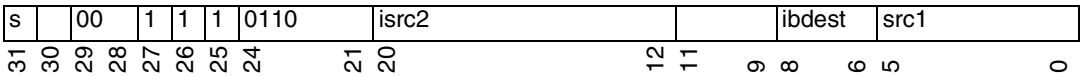
Description: Signed compare equal or less than

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmple Branch Register - Immediate
cmple $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 <= operand2;
B_{IBDEST} ←Bit(result1);

Description: Signed compare equal or less than

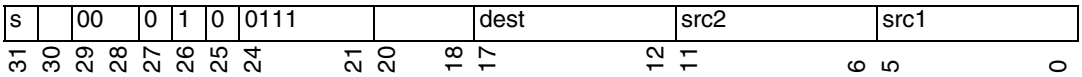
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpleu Register - Register

cmpleu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result1 ←operand1 <= operand2;
R_{DEST} ←Register(result1);

Description: Unsigned compare equal or less than

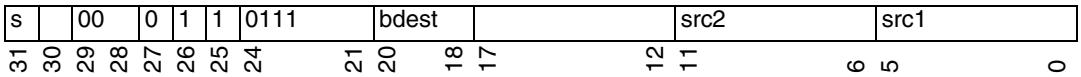
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpleu Branch Register - Register

cmpleu B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result1 ←operand1 <= operand2;
B_{BDEST} ←Bit(result1);

Description: Unsigned compare equal or less than

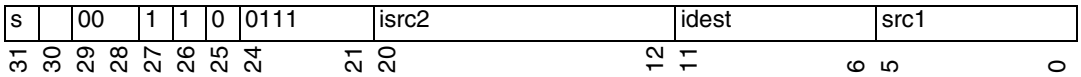
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpleu Register - Immediate

cmpleu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 <= operand2;
R_{IDEST} ←Register(result1);

Description: Unsigned compare equal or less than

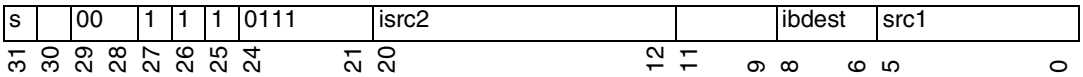
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpleu Branch Register - Immediate

cmpleu B_{IBDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 <= operand2;
B_{IBDEST} ←Bit(result1);

Description: Unsigned compare equal or less than

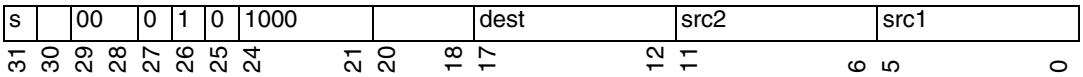
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmplt Register - Register

cmplt R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 < operand2;
R_{DEST} ←Register(result1);

Description: Signed compare less than

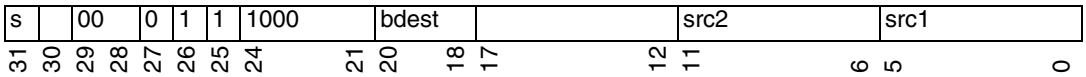
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmplt Branch Register - Register

cmplt $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 < operand2;

B_{BDEST} ←Bit(result1);

Description: Signed compare less than

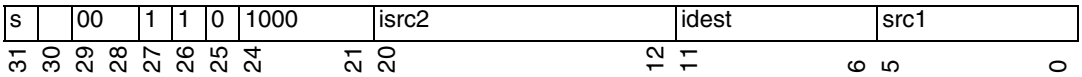
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmplt Register - Immediate

cmplt R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 < operand2;
R_{IDEST} ←Register(result1);

Description: Signed compare less than

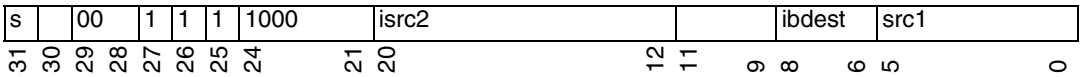
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmplt Branch Register - Immediate

cmplt B_{IBDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 < operand2;
B_{IBDEST} ←Bit(result1);

Description: Signed compare less than

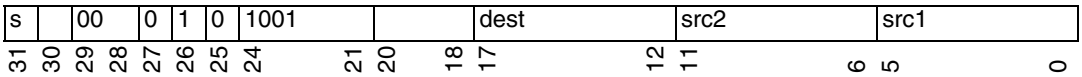
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpltu Register - Register

cmpltu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←operand1 < operand2;
```

```
RDEST ←Register(result1);
```

Description: Unsigned compare less than

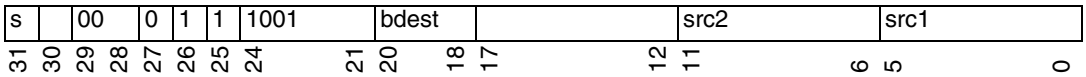
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpltu Branch Register - Register

cmpltu B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←operand1 < operand2;

B_{BDEST} ←Bit(result1);

Description: Unsigned compare less than

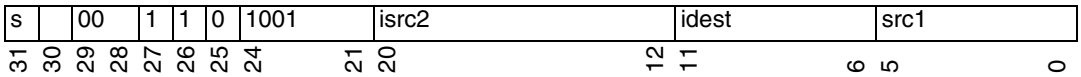
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpltu Register - Immediate

cmpltu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 < operand2;
R_{IDEST} ←Register(result1);

Description: Unsigned compare less than

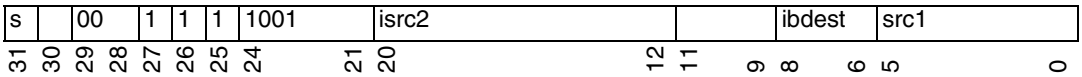
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpltu Branch Register - Immediate

cmpltu B_{IBDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 < operand2;
B_{IBDEST} ←Bit(result1);

Description: Unsigned compare less than

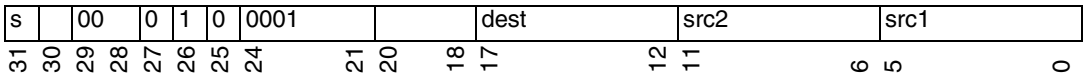
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpne Register - Register

cmpne $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←operand1 ! = operand2;
RDEST ←Register(result1);
```

Description: Test for inequality

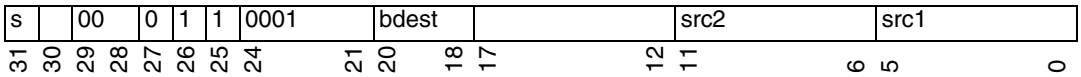
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpne Branch Register - Register

cmpne $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand1 ! = operand2;
B_{BDEST} ←Bit(result1);

Description: Test for inequality

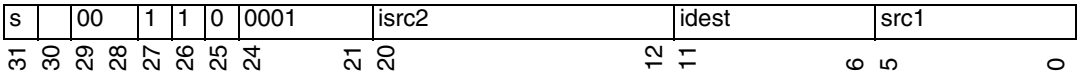
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpne Register - Immediate

cmpne $R_{IDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 ! = operand2;
R_{IDEST} ←Register(result1);

Description: Test for inequality

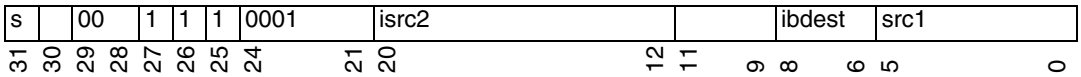
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

cmpne Branch Register - Immediate

cmpne $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ! = operand2;

B_{IBDEST} ←Bit(result1);

Description: Test for inequality

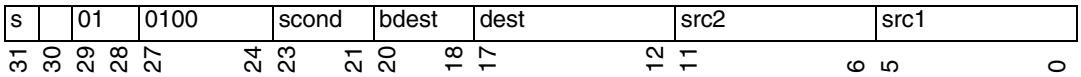
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

divs

divs R_{DEST}, B_{BDEST} = R_{SRC1}, R_{SRC2}, B_{SCOND}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
operand3 ←ZeroExtend$_{1}$(B_{SCOND});
temp ←ZeroExtend$_{32}$(operand1 × 2) ∨ (operand3 ∧ 1);
IF (operand1 < 0)
{
result1 ←temp + operand2;
result2 ←1;
}
ELSE
{
result1 ←temp - operand2;
result2 ←0;
}

R_{DEST} ←Register(result1);
B_{BDEST} ←Bit(result2);

Description: Non-restoring divide stage

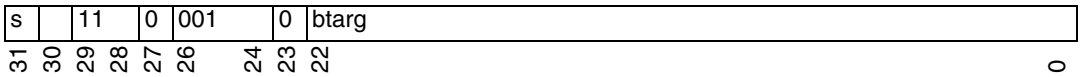
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

goto Immediate

goto BTARG



Semantics:

operand1 ←SignExtend$_{23}$(BTARG)<< 2;
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + operand1);

Description: Jump

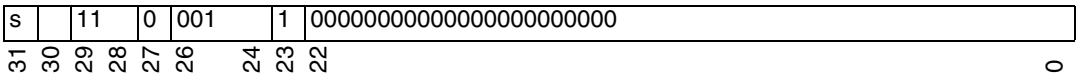
Restrictions: Must be the first syllable of a bundle.

No latency constraints.

Exceptions: None.

goto Link Register

goto $r63



Semantics:

PC ←Register(ZeroExtend$_{32}$(LR));

Description: Jump (using Link Register)

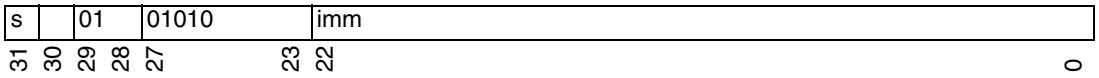
Restrictions: Must be the first syllable of a bundle.

There are no latency constraints between an call uprating the LR and this operation. There is a latency of 4 bundles between a load writing to the LR and this operation. There is a latency of 2 bundles between any other operation updating the LR and this operation.

Exceptions: None.

imml

imml IMM



Semantics:

extension ←ZeroExtend$_{23}$(IMM);

Description: Long immediate for previous syllable

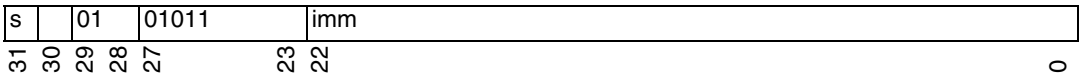
Restrictions: Must be encoded at an even word address.

No latency constraints.

Exceptions: None.

immr

immr IMM



Semantics:

extension ←ZeroExtend$_{23}$(IMM);

Description: Long immediate for next syllable

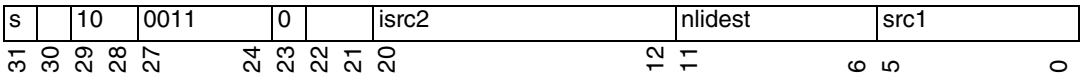
Restrictions: Must be encoded at an even word address.

No latency constraints.

Exceptions: None.

ldb

ldb $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
THROW CREG_ACCESS_VIOLATION;
ELSE
ReadCheckMemory8(ea);
```

```
ReadMemory8(ea);
result1 ←SignExtend8(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Signed load byte

Restrictions: Cannot write the link register ($r63).

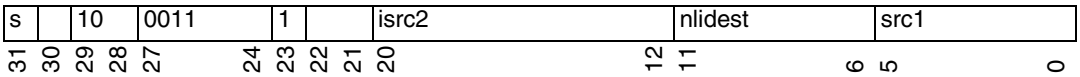
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

ldb.d

ldb.d $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (NOT (IsCRegSpace(ea)))
DisReadCheckMemory8(ea);
```

```
IF (NOT (IsCRegSpace(ea)))
DisReadMemory8(ea);
IF (IsCRegSpace(ea))
result1 ←0;
ELSE
result1 ←SignExtend8(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Signed load byte dismissable

Restrictions: Cannot write the link register ($r63).

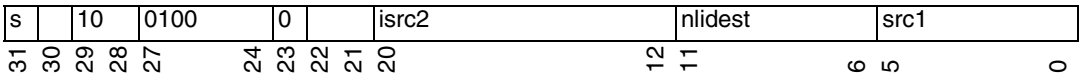
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, DTLB

ldbu

ldbu $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

operand1 ←SignExtend$_{32}$(Imm(ISRC2));
operand2 ←SignExtend$_{32}$(R_{SRC1});
ea ←ZeroExtend$_{32}$(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
THROW CREG_ACCESS_VIOLATION;
ELSE
ReadCheckMemory$_8$(ea);

ReadMemory$_8$(ea);
result1 ←ZeroExtend$_8$(ReadMemResponse());
$R_{NLIDEST}$ ←Register(result1);

Description: Unsigned load byte

Restrictions Cannot write the link register ($r63).

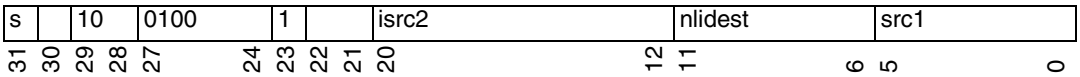
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

ldbu.d

ldbu.d $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (NOT (IsCRegSpace(ea)))
DisReadCheckMemory8(ea);
```

```
IF (NOT (IsCRegSpace(ea)))
DisReadMemory8(ea);
IF (IsCRegSpace(ea))
result1 ←0;
ELSE
result1 ←ZeroExtend8(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Unsigned load byte dismissable

Restrictions: Cannot write the link register ($r63).

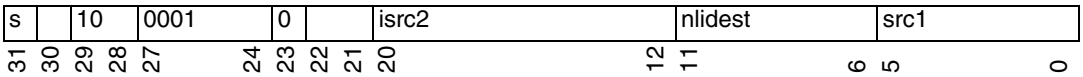
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, DTLB

ldh

ldh $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
THROW CREG_ACCESS_VIOLATION;
ELSE
ReadCheckMemory16(ea);
```

```
ReadMemory16(ea);
result1 ←SignExtend16(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Signed load half-word

Restrictions: Cannot write the link register ($r63).

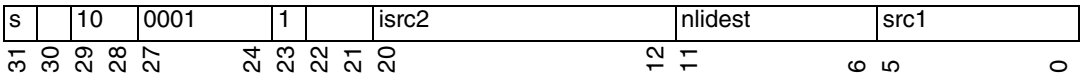
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldh.d

ldh.d $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (NOT (IsCRegSpace(ea)))
DisReadCheckMemory16(ea);
```

```
IF (NOT (IsCRegSpace(ea)))
DisReadMemory16(ea);
IF (IsCRegSpace(ea))
result1 ←0;
ELSE
result1 ←SignExtend16(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Signed load half-word dismissable

Restrictions: Cannot write the link register ($r63).

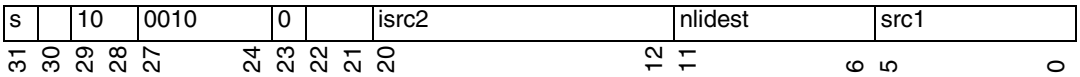
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, DTLB, MISALIGNED_TRAP

ldhu

ldhu $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

operand1 ←SignExtend$_{32}$(Imm(ISRC2));
operand2 ←SignExtend$_{32}$(R_{SRC1});
ea ←ZeroExtend$_{32}$(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
THROW CREG_ACCESS_VIOLATION;
ELSE
ReadCheckMemory$_{16}$(ea);

ReadMemory$_{16}$(ea);
result1 ←ZeroExtend$_{16}$(ReadMemResponse());
$R_{NLIDEST}$ ←Register(result1);

Description: Unsigned load half-word

Restrictions: Cannot write the link register ($r63).

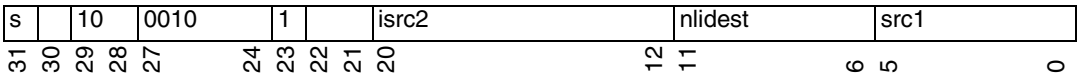
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldhu.d

ldhu.d $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (NOT (IsCRegSpace(ea)))
DisReadCheckMemory16(ea);
```

```
IF (NOT (IsCRegSpace(ea)))
DisReadMemory16(ea);
IF (IsCRegSpace(ea))
result1 ←0;
ELSE
result1 ←ZeroExtend16(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Unsigned load half-word dismissable

Restrictions: Cannot write the link register ($r63).

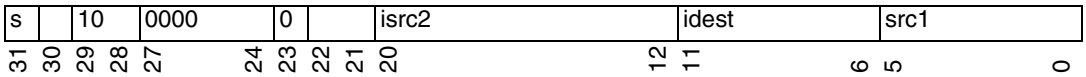
Uses the ld/st unit for which only one operation is allowed per bundle.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, DTLB, MISALIGNED_TRAP

ldw

ldw R_{IDEST} = ISRC2[R_{SRC1}]



Semantics:

operand1 ←SignExtend$_{32}$(Imm(ISRC2));
operand2 ←SignExtend$_{32}$(R_{SRC1});
ea ←ZeroExtend$_{32}$(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
ReadCheckCReg(ea);
ELSE
ReadCheckMemory$_{32}$(ea);

IF (IsCRegSpace(ea))
ReadCReg(ea);
ELSE
ReadMemory$_{32}$(ea);
result1 ←SignExtend$_{32}$(ReadMemResponse());
R_{IDEST} ←Register(result1);

Description: Load word

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

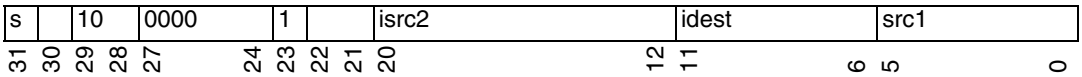
There is a latency of 2 bundles before R_{IDEST} is available for reading.

If writing the LR ($r63), there is a latency of 3 bundles before a call LR or goto LR is issued.

Exceptions: DBREAK, DTLB, MISALIGNED_TRAP, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING

ldw.d

ldw.d R_{IDEST} = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (NOT (IsCRegSpace(ea)))
DisReadCheckMemory32(ea);
```

```
IF (NOT (IsCRegSpace(ea)))
DisReadMemory32(ea);
IF (IsCRegSpace(ea))
result1 ←0;
ELSE
result1 ←SignExtend32(ReadMemResponse());
RIDEST ←Register(result1);
```

Description: Load word dismissable

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

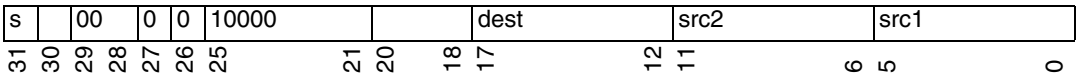
There is a latency of 2 bundles before R_{IDEST} is available for reading.

If writing the LR ($r63), there is a latency of 3 bundles before a call LR or goto LR is issued.

Exceptions: DBREAK, DTLB, MISALIGNED_TRAP, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING

max Register

max $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
IF (operand1 > operand2)
result1 ←operand1;
ELSE
result1 ←operand2;
```

```
RDEST ←Register(result1);
```

Description: Signed maximum

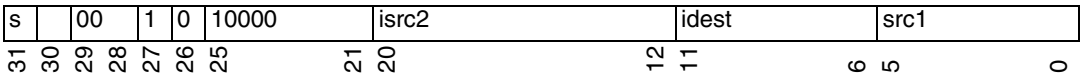
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

max Immediate

max $R_{IDEST} = R_{SRC1}$, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
IF (operand1 > operand2)
result1 ←operand1;
ELSE
result1 ←operand2;
```

```
RIDEST ←Register(result1);
```

Description: Signed maximum

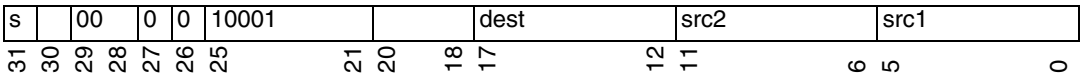
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

maxu Register

maxu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
IF (operand1 > operand2)
result1 ←operand1;
ELSE
result1 ←operand2;
```

```
RDEST ←Register(result1);
```

Description: Unsigned maximum

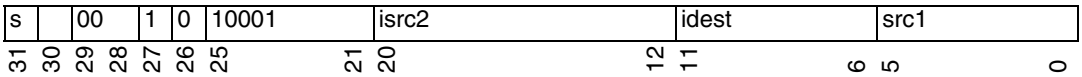
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

maxu Immediate

maxu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
IF (operand1 > operand2)
result1 ←operand1;
ELSE
result1 ←operand2;

R_{IDEST} ←Register(result1);

Description: Unsigned maximum

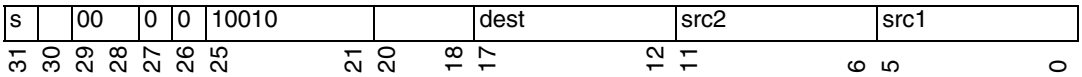
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

min Register

min R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
IF (operand1 < operand2)
result1 ←operand1;
ELSE
result1 ←operand2;

R_{DEST} ←Register(result1);

Description: Signed minimum

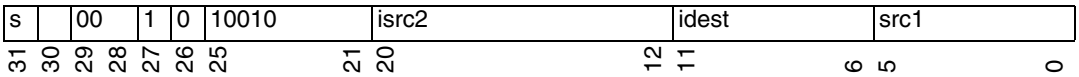
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

min Immediate

min $R_{IDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
IF (operand1 < operand2)
result1 ←operand1;
ELSE
result1 ←operand2;

R_{IDEST} ←Register(result1);

Description: Signed minimum

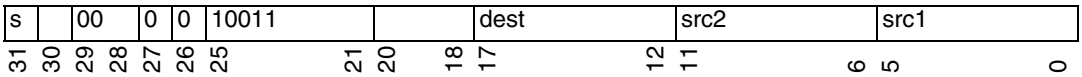
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

minu Register

minu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
IF (operand1 < operand2)
result1 ←operand1;
ELSE
result1 ←operand2;
```

```
RDEST ←Register(result1);
```

Description: Unsigned minimum

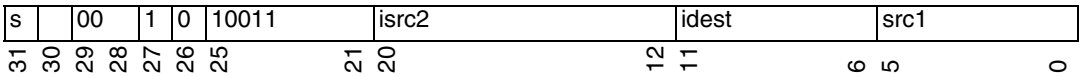
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

minu Immediate

minu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(Imm(ISRC2));
IF (operand1 < operand2)
result1 ←operand1;
ELSE
result1 ←operand2;
```

```
RIDEST ←Register(result1);
```

Description: Unsigned minimum

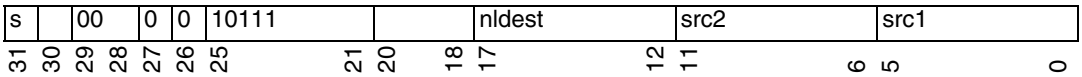
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

mulh Register

mulh R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 × (operand2 >> 16); R_{NLDEST} ←Register(result1);

Description: Word by upper-half-word signed multiply

Restrictions: Cannot write the link register ($r63).

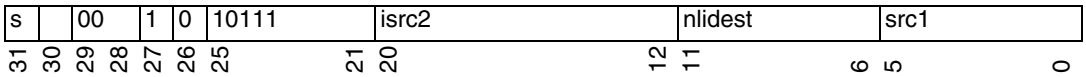
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulh Immediate

mulh $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 × (operand2 >> 16); $R_{NLIDEST}$ ←Register(result1);

Description: Word by upper-half-word signed multiply

Restrictions: Cannot write the link register ($r63).

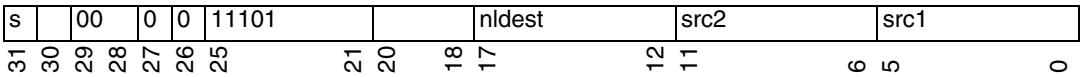
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulhh Register

mulhh R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 >> 16) × (operand2 >> 16); R_{NLDEST} ←Register(result1);

Description: Upper-half-word by upper-half-word signed multiply

Restrictions: Cannot write the link register ($r63).

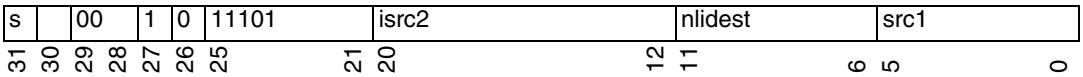
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhh Immediate

mulhh $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 >> 16) × (operand2 >> 16); $R_{NLIDEST}$ ←Register(result1);

Description: Upper-half-word by upper-half-word signed multiply

Restrictions: Cannot write the link register ($r63).

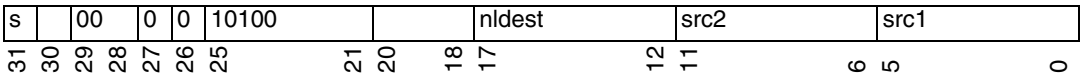
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulhhs Register

mulhhs R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 × (operand2 >> 16)) >> 16; R_{NLDEST} ←Register(result1);

Description: Word by upper-half-word signed multiply, returns top 32 bits of 48 bit result

Restrictions: Cannot write the link register ($r63).

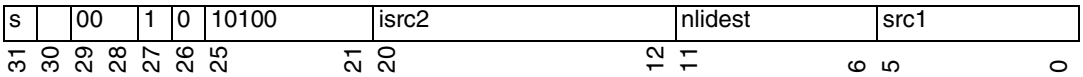
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhhs Immediate

mulhhs $R_{NLIDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 × (operand2 >> 16)) >> 16; $R_{NLIDEST}$ ←Register(result1);

Description: Word by upper-half-word signed multiply, returns top 32 bits of 48 bit result

Restrictions: Cannot write the link register ($r63).

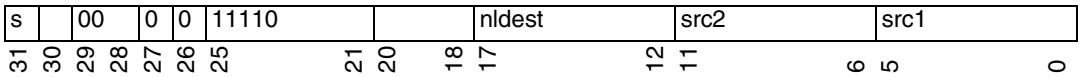
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulhhu Register

mulhhu R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←ZeroExtend$_{16}$(operand1 >> 16) × ZeroExtend$_{16}$(operand2 >> 16); R_{NLDEST} ←Register(result1);

Description: Upper-half-word by upper-half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

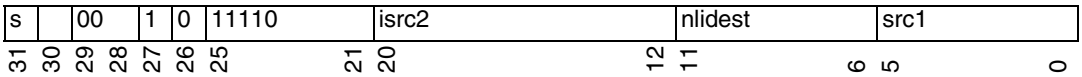
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhhu Immediate

mulhhu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←ZeroExtend$_{16}$(operand1 >> 16) × ZeroExtend$_{16}$(operand2 >> 16); $R_{NLIDEST}$ ←Register(result1);

Description: Upper-half-word by upper-half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

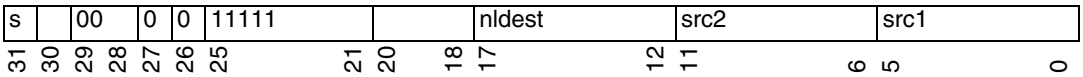
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulhs Register

mulhs R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 × ZeroExtend$_{16}$(operand2 >> 16)) << 16; R_{NLDEST} ←Register(result1);

Description: Sign extended word by zero extended upper-half-word signed multiply. Result is left shifted 16 places.

Restrictions: Cannot write the link register ($r63).

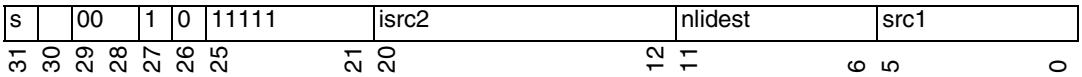
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhs Immediate

mulhs $R_{NLIDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 × ZeroExtend$_{16}$(operand2 >> 16)) << 16; $R_{NLIDEST}$ ←Register(result1);

Description: Sign extended word by zero extended upper-half-word signed multiply. Result is left shifted 16 places.

Restrictions: Cannot write the link register ($r63).

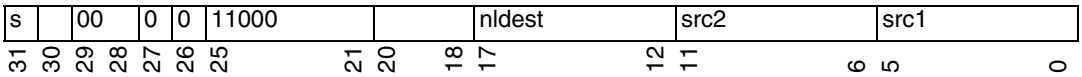
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulhu Register

mulhu $R_{NLDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); R_{NLDEST} ←Register(result1);

Description: Word by upper-half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

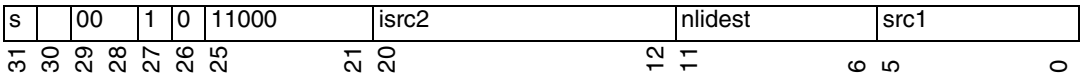
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhu Immediate

mulhu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));

result1 ←operand1 × ZeroExtend$_{16}$(operand2 >> 16);
$R_{NLIDEST}$ ←Register(result1);

Description: Word by upper-half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

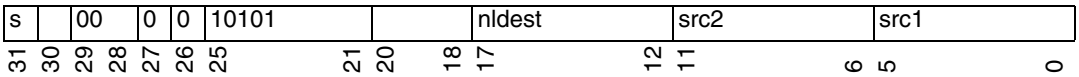
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mull Register

mull R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(R_{SRC2});
result1 ←operand1 × operand2; R_{NLDEST} ←Register(result1);

Description: Word by half-word signed multiply

Restrictions: Cannot write the link register ($r63).

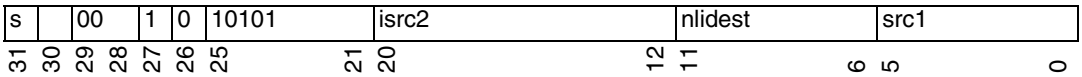
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mull Immediate

mull $R_{NLIDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(Imm(ISRC2));
result1 ←operand1 × operand2; $R_{NLIDEST}$ ←Register(result1);

Description: Word by half-word signed multiply

Restrictions: Cannot write the link register ($r63).

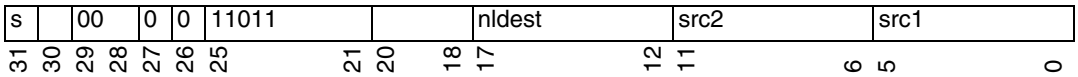
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mullh Register

mullh R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 × (operand2 >> 16); R_{NLDEST} ←Register(result1);

Description: Half-word by upper-half-word signed multiply

Restrictions: Cannot write the link register ($r63).

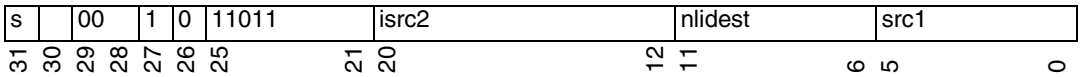
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mullh Immediate

mullh $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 × (operand2 >> 16); $R_{NLIDEST}$ ←Register(result1);

Description: Half-word by upper-half-word signed multiply

Restrictions: Cannot write the link register ($r63).

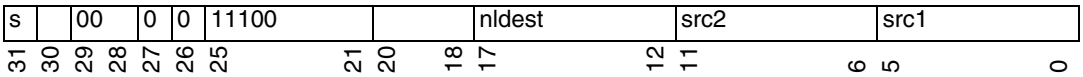
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mullhu Register

mullhu R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); R_{NLDEST} ←Register(result1);

Description: Half-word by upper-half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

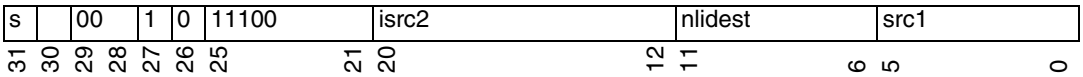
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mullhu Immediate

mullhu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); $R_{NLIDEST}$ ←Register(result1);

Description: Half-word by upper-half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

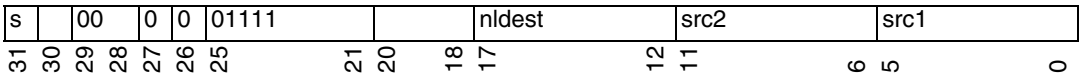
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mullhus Register

mullhus $R_{NLDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(R_{SRC2});
result1 ←(operand1 × operand2) >> 32; R_{NLDEST} ←Register(result1);

Description: Sign extended word by zero extended lower-half-word signed multiply. Returns top 16 bits of 48 bit result, sign extended.

Restrictions: Cannot write the link register ($r63).

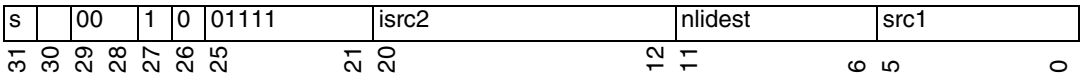
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mullhus Immediate

mullhus $R_{NLIDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(Imm(ISRC2));
result1 ←(operand1 × operand2) >> 32; $R_{NLIDEST}$ ←Register(result1);

Description: Sign extended word by zero extended lower-half-word signed multiply. Returns top 16 bits of 48 bit result, sign extended.

Restrictions: Cannot write the link register ($r63).

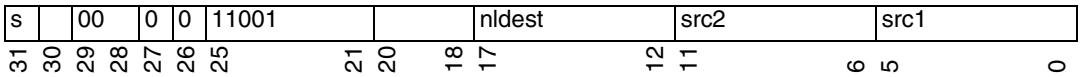
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulll Register

mulll R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(R_{SRC2});
result1 ←operand1 × operand2; R_{NLDEST} ←Register(result1);

Description: Half-word by half-word signed multiply

Restrictions: Cannot write the link register ($r63).

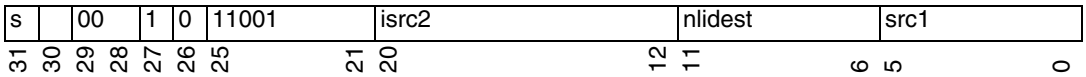
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulll Immediate

mulll $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(Imm(ISRC2));
result1 ←operand1 × operand2; $R_{NLIDEST}$ ←Register(result1);

Description: Half-word by half-word signed multiply

Restrictions: Cannot write the link register ($r63).

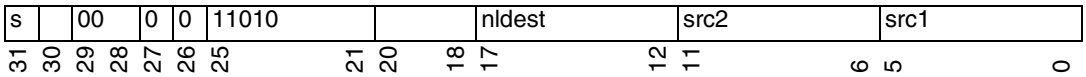
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulllu Register

mulllu R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(R_{SRC2});
result1 ←operand1 × operand2; R_{NLDEST} ←Register(result1);

Description: Half-word by half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

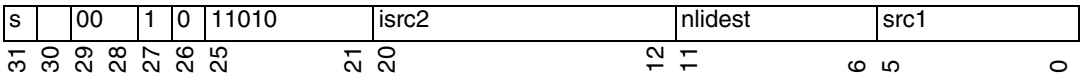
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulllu Immediate

mulllu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(Imm(ISRC2));
result1 ←operand1 × operand2; $R_{NLIDEST}$ ←Register(result1);

Description: Half-word by half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

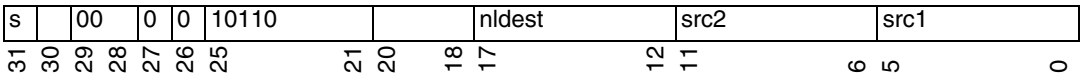
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mullu Register

mullu R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(R_{SRC2});
result1 ←operand1 × operand2; R_{NLDEST} ←Register(result1);

Description: Word by half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

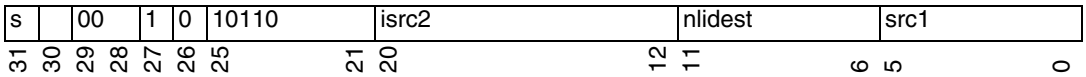
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mullu Immediate

mullu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(Imm(ISRC2));
result1 ←operand1 × operand2; $R_{NLIDEST}$ ←Register(result1);

Description: Word by half-word unsigned multiply

Restrictions: Cannot write the link register ($r63).

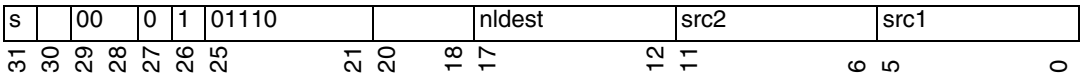
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mul32 Register

mul32 R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 × operand2; R_{NLDEST} ←Register(result1);

Description: 32 by 32 multiply

Restrictions: Cannot write the link register ($r63).

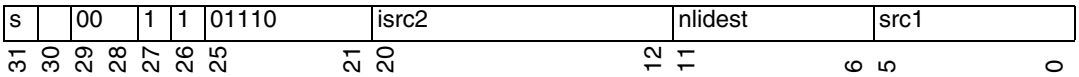
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mul32 Immediate

mul32 $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 × operand2; $R_{NLIDEST}$ ←Register(result1);

Description: 32 by 32 multiply

Restrictions: Cannot write the link register ($r63).

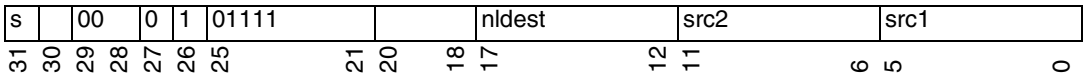
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mul64h Register

mul64h R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 × operand2) >> 32; R_{NLDEST} ←Register(result1);

Description: 32 by 32 signed multiply, return the top 32 bits of the 64 bit result.

Restrictions: Cannot write the link register ($r63).

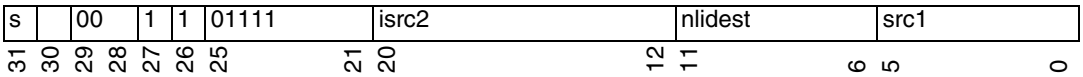
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mul64h Immediate

mul64h $R_{NLIDEST}$ = R_{SRC1}, ISRC2



operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));

result1 ←(operand1 × operand2) >> 32;
$R_{NLIDEST}$ ←Register(result1);

Description: 32 by 32 signed multiply, return the top 32 bits of the 64 bit result.

Restrictions: Cannot write the link register ($r63).

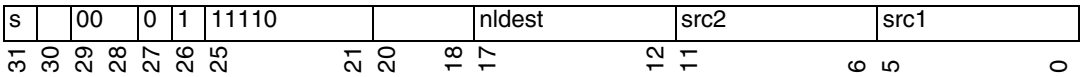
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mul64hu Register

mul64hu $R_{NLDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←(operand1 × operand2) >> 32; R_{NLDEST} ←Register(result1);

Description: 32 by 32 unsigned multiply, return the top 32 bits of the 64 bit result.

Restrictions: Cannot write the link register ($r63).

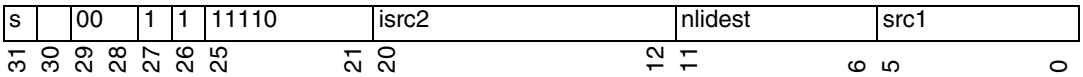
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mul64hu Immediate

mul64hu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 × operand2) >> 32; $R_{NLIDEST}$ ←Register(result1);

Description: 32 by 32 unsigned multiply, return the top 32 bits of the 64 bit result.

Restrictions: Cannot write the link register ($r63).

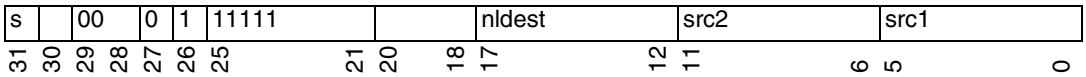
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulfrac Register

mulfrac R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
IF (((-operand1) = 0x80000000) AND ((-operand2) = 0x80000000))
{
result1 ←0x7FFFFFFF;
}
ELSE
{
result1 ←operand1 × operand2;
result1 ←result1 + (1 << 30);
result1 ←result1 >> 31;
}
RNLDEST ←Register(result1);
```

Description: fractional multiply.

Restrictions: Cannot write the link register ($r63).

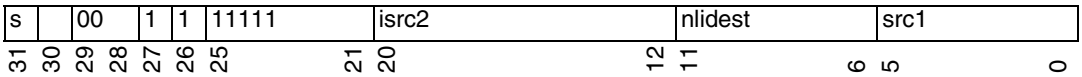
Must be encoded at an odd word address.

There is a latency of 2 bundles before R_{NLDEST} is available for reading.

Exceptions: None.

mulfrac Immediate

mulfrac $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
IF (((-operand1) = 0x80000000) AND ((-operand2) = 0x80000000))
{
result1 ←0x7FFFFFFF;
}
ELSE
{
result1 ←operand1 × operand2;
result1 ←result1 + (1 << 30);
result1 ←result1 >> 31;
}
RNLIDEST ←Register(result1);
```

Description: fractional multiply.

Restrictions: Cannot write the link register ($r63).

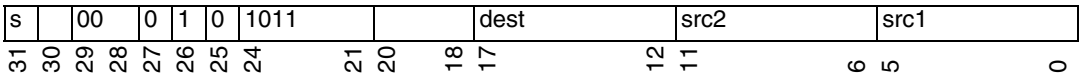
Must be encoded at an odd word address.

There is a latency of 2 bundles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

nandl Register - Register

nandl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));
R_{DEST} ←Register(result1);

Description: Logical nand

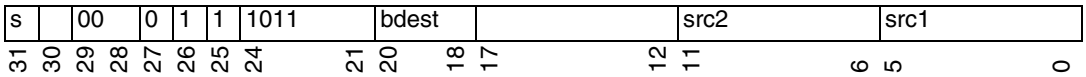
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

nandl Branch Register - Register

nandl $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));

B_{BDEST} ←Bit(result1);

Description: Logical nand

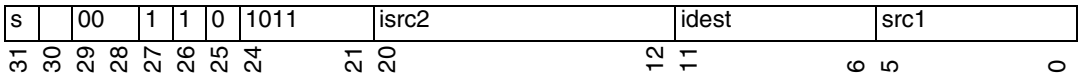
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

nandl Register - Immediate

nandl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));
```

```
RIDEST ←Register(result1);
```

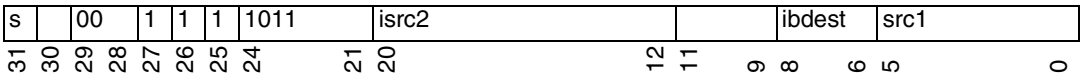
Description: Logical nand

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

nandl Branch Register - Immediate
nandl B_{IBDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));

B_{IBDEST} ←Bit(result1);

Description: Logical nand

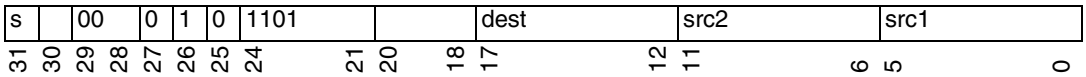
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

norl Register - Register

norl $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));

R_{DEST} ←Register(result1);

Description: Logical nor

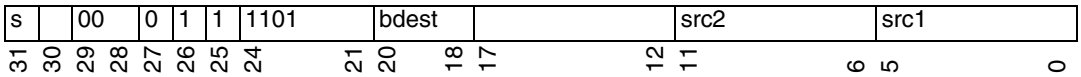
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

norl Branch Register - Register

norl $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));

B_{BDEST} ←Bit(result1);

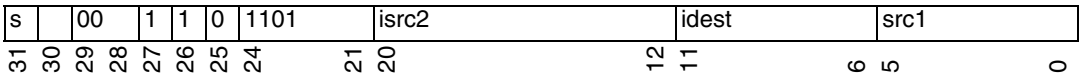
Description: Logical nor

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

norl Register - Immediate
norl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));

R_{IDEST} ←Register(result1);

Description: Logical nor

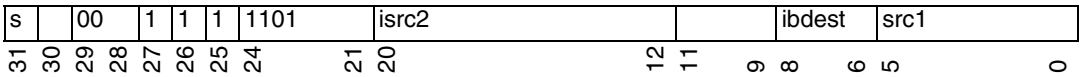
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

norl Branch Register - Immediate

norl $B_{IBDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));

B_{IBDEST} ←Bit(result1);

Description: Logical nor

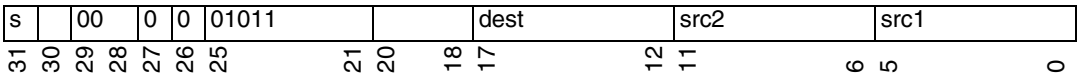
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

or Register

or $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ∨ operand2;

R_{DEST} ←Register(result1);

Description: Bitwise or

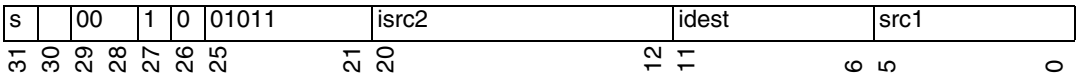
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

or Immediate

or $R_{IDEST} = R_{SRC1}$, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ∨ operand2;

R_{IDEST} ←Register(result1);

Description: Bitwise or

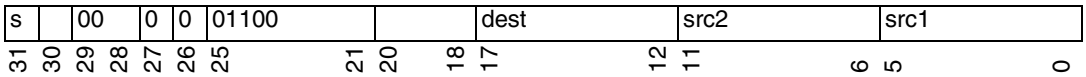
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

orc Register

orc $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←(~operand1) ∨  operand2;
```

```
RDEST ←Register(result1);
```

Description: Complement and bitwise or

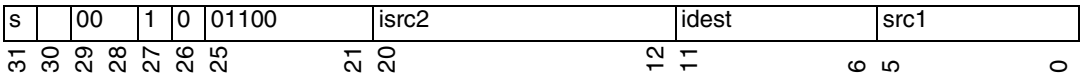
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

orc Immediate

orc R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
result1 ←(~operand1) ∨  operand2;
```

```
RIDEST ←Register(result1);
```

Description: Complement and bitwise or

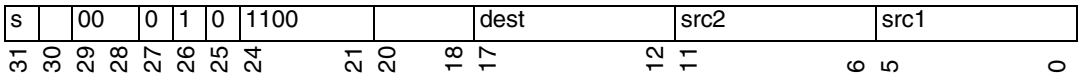
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

orl Register - Register

orl $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);

R_{DEST} ←Register(result1);

Description: Logical or

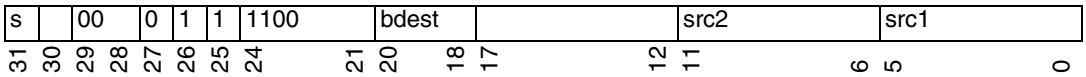
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

orl Branch Register - Register

orl $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);

B_{BDEST} ←Bit(result1);

Description: Logical or

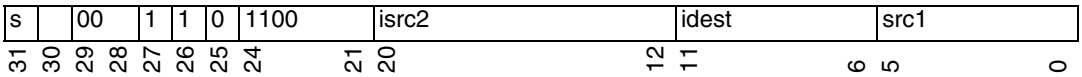
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

orl Register - Immediate

orl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);
```

```
RIDEST ←Register(result1);
```

Description: Logical or

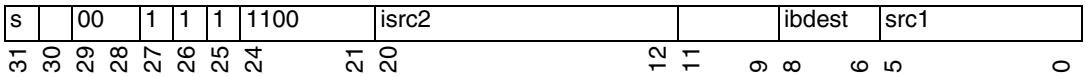
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

orl Branch Register - Immediate

orl B_{IBDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);

B_{IBDEST} ←Bit(result1);

Description: Logical or

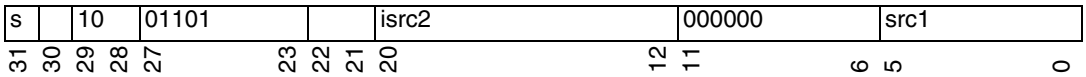
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

pft

pft ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
PrefetchCheckMemory(ea);
```

```
PrefetchMemory(ea);
```

Description: Prefetch

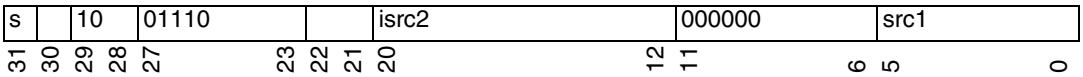
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: None.

prgadd

prgadd ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
PurgeAddressCheckMemory(ea);
```

```
PurgeAddress(ea);
```

Description: Purge the address given from the data cache

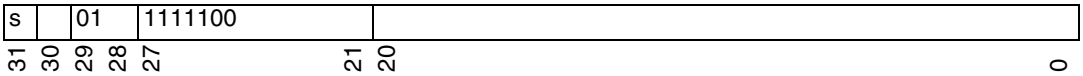
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: DTLB

prgins

prgins



Semantics:

```
IF (PSW[USER_MODE])
THROW ILL_INST;
```

```
PurgeIns();
```

Description: Purge the instruction cache

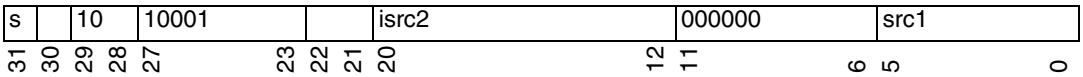
Restrictions: Must be in a bundle by itself

No latency constraints.

Exceptions: ILL_INST

prginspg

prginspg ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
IF (PSW[USER_MODE])
THROW ILL_INST;
ea ←ZeroExtend32(operand1 + operand2);
```

```
PurgeInsPg(ea);
```

Description: Purge a 8kb page from the instruction cache

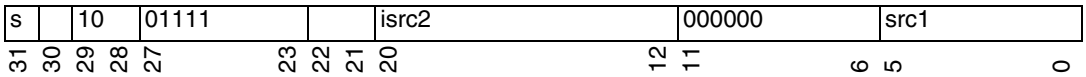
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: ILL_INST

prgset

prgset ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
```

```
PurgeSet(ea);
```

Description: Purge a set of four cache lines from the data cache

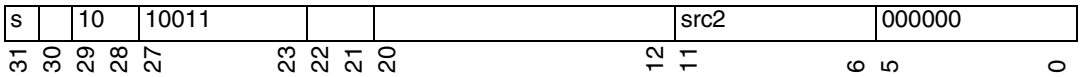
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: None.

pswclr

pswclr R_{SRC2}



Semantics:

operand2 ←SignExtend$_{32}$(R_{SRC2}); IF (PSW[USER_MODE]) THROW ILL_INST;
PswClr(operand2);

Description: Atomic psw clear.

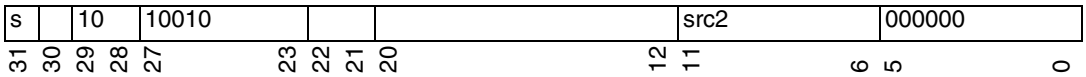
Restrictions: Must be the first in a bundle and uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: ILL_INST

pswset

pswset R_{SRC2}



Semantics:

operand2 ←SignExtend$_{32}$(R_{SRC2}); IF (PSW[USER_MODE]) THROW ILL_INST;
PswSet(operand2);

Description: Atomic PSW set.

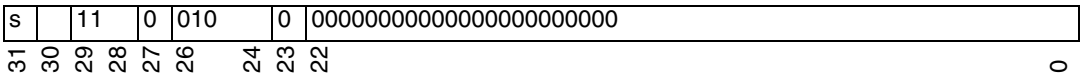
Restrictions: Must be the first in a bundle and uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: ILL_INST

rfi

rfi



Semantics:

```
IF (PSW[USER_MODE])
THROW ILL_INST;
PC ←Register(ZeroExtend32(SAVED_PC));
```

```
PSW ←SAVED_PSW;
SAVED_PC←SAVED_SAVED_PC;
SAVED_PSW←SAVED_SAVED_PSW;
Rfi();
```

Description: Return from interrupt

Restrictions: Must be the first in a bundle and uses the ld/st unit for which only one operation is allowed per bundle.

Instructions writing SAVED_PC must be followed by 4 bundles before this instruction can be issued.

Instructions writing SAVED_PSW must be followed by 4 bundles before this instruction can be issued.

Instructions writing SAVED_SAVED_PC must be followed by 4 bundles before this instruction can be issued.

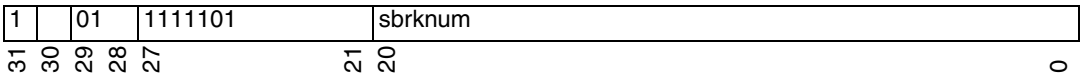
Instructions writing SAVED_SAVED_PSW must be followed by 4 bundles before this instruction can be issued.

Instructions writing PSW must be followed by 4 bundles before this instruction can be issued.

Exceptions: ILL_INST

sbrk

sbrk SBRKNUM



Semantics:

operand1 ←ZeroExtend$_{21}$(SBRKNUM); THROW SBREAK;

Description: Software breakpoint

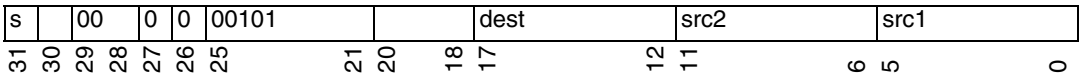
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: SBREAK

sh1add Register

sh1add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←(operand1 << 1) + operand2;
R_{DEST} ←Register(result1);

Description: Shift left one and accumulate

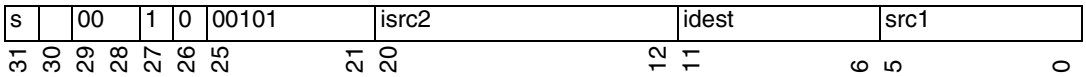
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh1add Immediate

sh1add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 << 1) + operand2;

R_{IDEST} ←Register(result1);

Description: Shift left one and accumulate

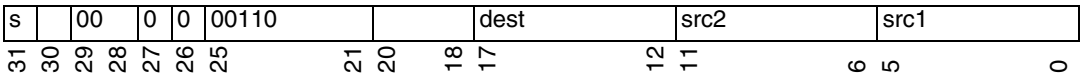
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh2add Register

sh2add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←(operand1 << 2) + operand2;
```

```
RDEST ←Register(result1);
```

Description: Shift left two and accumulate

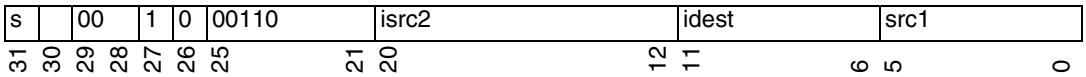
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh2add Immediate

sh2add R_{IDEST} = RS_{RC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 << 2) + operand2;

R_{IDEST} ←Register(result1);

Description: Shift left two and accumulate

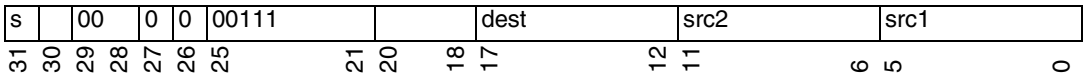
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh3add Register

sh3add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 << 3) + operand2;

R_{DEST} ←Register(result1);

Description: Shift left three and accumulate

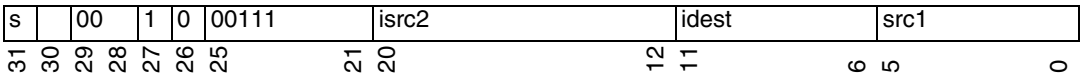
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh3add Immediate

sh3add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
result1 ←(operand1 << 3) + operand2;
```

```
RIDEST ←Register(result1);
```

Description: Shift left three and accumulate

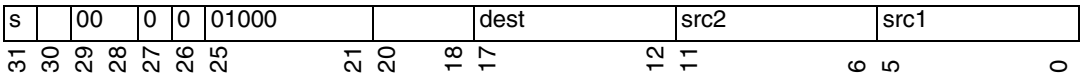
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh4add Register

sh4add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←(operand1 << 4) + operand2;
```

```
RDEST ←Register(result1);
```

Description: Shift left four and accumulate

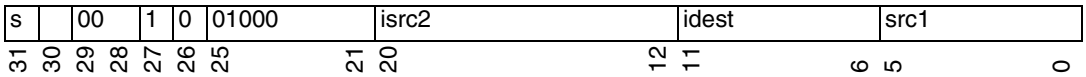
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sh4add Immediate

sh4add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
result1 ←(operand1 << 4) + operand2;
```

```
RIDEST ←Register(result1);
```

Description: Shift left four and accumulate

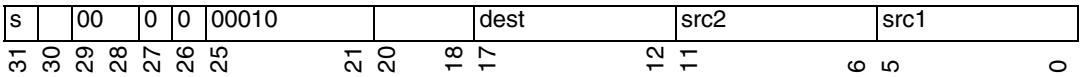
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

shl Register

shl $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 31)
result1 ←0;
ELSE
result1 ←operand1 << operand2;
```

```
RDEST ←Register(result1);
```

Description: Shift left

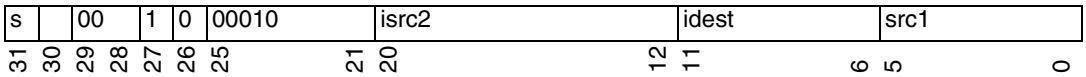
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

shl Immediate

shl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 31)
result1 ←0;
ELSE
result1 ←operand1 << operand2;
```

```
RIDEST ←Register(result1);
```

Description: Shift left

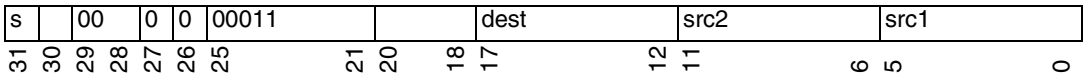
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

shr Register

shr $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{8}$(R_{SRC2});
result1 ←operand1 >> operand2;

R_{DEST} ←Register(result1);

Description: Arithmetic shift right

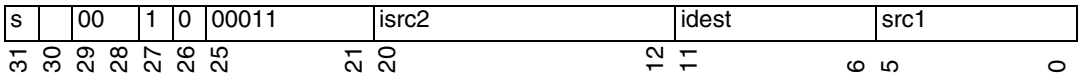
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

shr Immediate

shr R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{8}$(Imm(ISRC2));
result1 ←operand1 >> operand2;

R_{IDEST} ←Register(result1);

Description: Arithmetic shift right

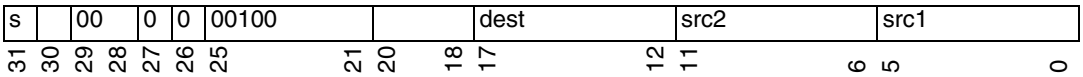
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

shru Register

shru R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{8}$(R_{SRC2}); result1 ←operand1 >> operand2;
R_{DEST} ←Register(result1);

Description: Logical shift right

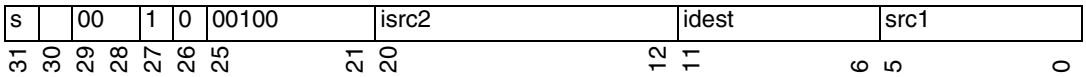
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

shru Immediate

shru R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{8}$(Imm(ISRC2)); result1 ←operand1 >> operand2;
R_{IDEST} ←Register(result1);

Description: Logical shift right

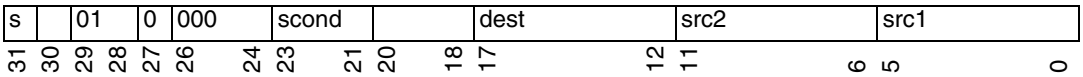
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

slct Register

slct $R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_1$(B_{SCOND});
operand2 ←SignExtend$_{32}$(R_{SRC1});
operand3 ←SignExtend$_{32}$(R_{SRC2});
IF (operand1 ≠ 0)
result1 ←operand2;
ELSE
result1 ←operand3;

R_{DEST} ←Register(result1);

Description: Conditional select

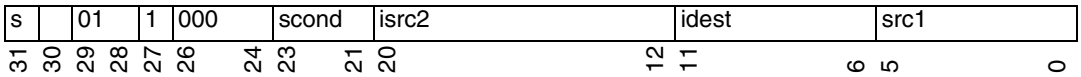
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

slct Immediate

slct $R_{IDEST} = B_{SCOND}$, R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend1(B_SCOND);
operand2 ←SignExtend32(R_SRC1);
operand3 ←SignExtend32(Imm(ISRC2));
IF (operand1 ≠ 0)
result1 ←operand2;
ELSE
result1 ←operand3;
```

```
R_IDEST ←Register(result1);
```

Description: Conditional select

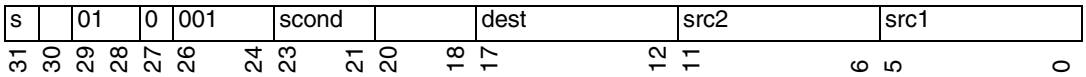
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

slctf Register

slctf $R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_1$(B_{SCOND});
operand2 ←SignExtend$_{32}$(R_{SRC1});
operand3 ←SignExtend$_{32}$(R_{SRC2});
IF (operand1 = 0)
result1 ←operand2;
ELSE
result1 ←operand3;

R_{DEST} ←Register(result1);

Description: Conditional select

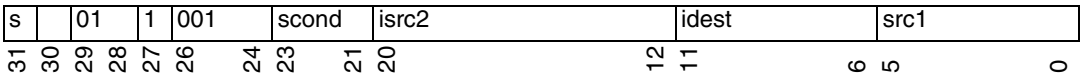
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

slctf Immediate

slctf $R_{IDEST} = B_{SCOND}$, R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend1(BSCOND);
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(Imm(ISRC2));
IF (operand1 = 0)
result1 ←operand2;
ELSE
result1 ←operand3;
```

```
RIDEST ←Register(result1);
```

Description: Conditional select

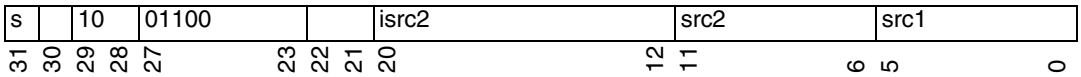
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

stb

stb ISRC2[R_{SRC1}] = R_{SRC2}



Semantics:

operand1 ←$SignExtend_{32}$(Imm(ISRC2));
operand2 ←$SignExtend_{32}$(R_{SRC1});
operand3 ←$SignExtend_{32}$(R_{SRC2});
ea ←$ZeroExtend_{32}$(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
THROW CREG_ACCESS_VIOLATION;
$WriteCheckMemory_8$(ea);

$WriteMemory_8$(ea, operand3);

Description: Store byte

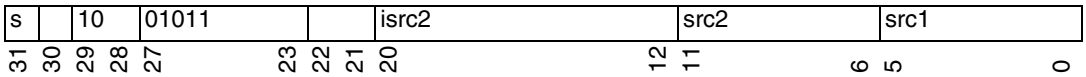
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

sth

sth ISRC2[R_{SRC1}] = R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
THROW CREG_ACCESS_VIOLATION;
WriteCheckMemory16(ea);
```

```
WriteMemory16(ea, operand3);
```

Description: Store half-word

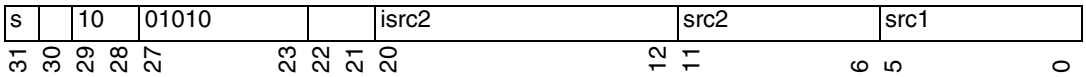
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

stw

stw ISRC2[R_{SRC1}] = R_{SRC2}



Semantics:

operand1 ←$SignExtend_{32}$(Imm(ISRC2));
operand2 ←$SignExtend_{32}$(R_{SRC1});
operand3 ←$SignExtend_{32}$(R_{SRC2});
ea ←$ZeroExtend_{32}$(operand1 + operand2);
IF (IsDBreakHit(ea))
THROW DBREAK;
IF (IsCRegSpace(ea))
WriteCheckCReg(ea);
ELSE
$WriteCheckMemory_{32}$(ea);

IF (IsCRegSpace(ea))
WriteCReg(ea, operand3);
ELSE
$WriteMemory_{32}$(ea, operand3);

Description: Store word

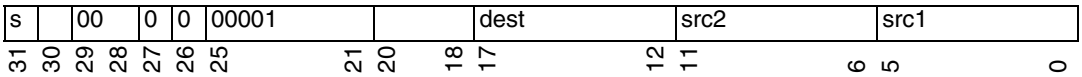
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: DBREAK, DTLB, MISALIGNED_TRAP, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING

sub Register

sub R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result1 ←operand2 - operand1;
R_{DEST} ←Register(result1);

Description: Subtract

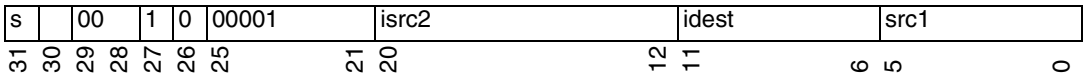
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sub Immediate

sub R_{IDEST} = ISRC2, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand2 - operand1;
R_{IDEST} ←Register(result1);

Description: Subtract

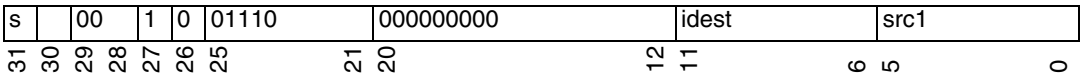
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sxtb

sxtb $R_{IDEST} = R_{SRC1}$



Semantics:

operand1 ←SignExtend$_8$(R_{SRC1}); result1 ←operand1;
R_{IDEST} ←Register(result1);

Description: Sign extend byte

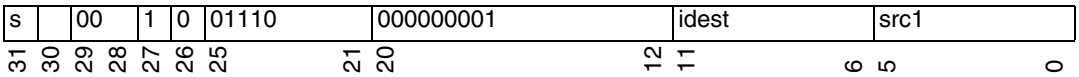
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sxth

sxth $R_{IDEST} = R_{SRC1}$



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); result1 ←operand1;
R_{IDEST} ←Register(result1);

Description: Sign extend half

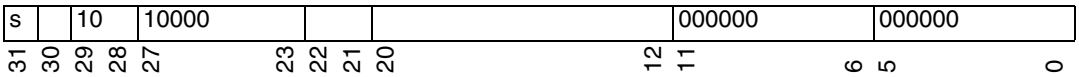
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

sync

sync



Semantics:

Sync();

Description: Ensure synchronization

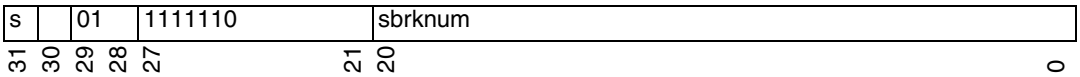
Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.

No latency constraints.

Exceptions: None.

syscall

syscall SBRKNUM



Semantics:

operand1 ←ZeroExtend$_{21}$(SBRKNUM); THROW SYSCALL;

Description: System call

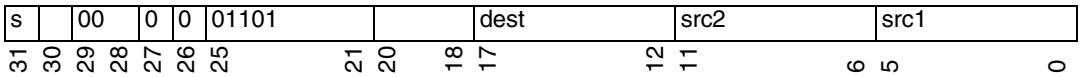
Restrictions: Must be in a bundle by itself

No latency constraints.

Exceptions: SYSCALL

xor Register

xor $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ⊕ operand2;

R_{DEST} ←Register(result1);

Description: Bitwise exclusive-or

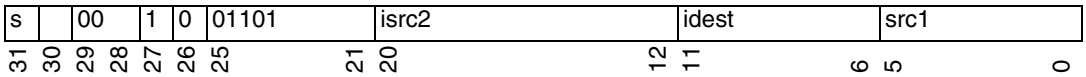
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

xor Immediate

xor R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(ISRC2)); result1 ←operand1 ⊕ operand2;
R_{IDEST} ←Register(result1);

Description: Bitwise exclusive-or

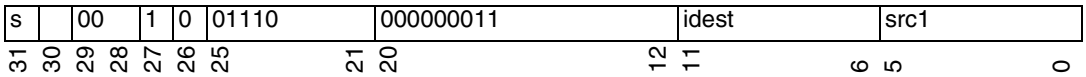
Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

zxth

zxth $R_{IDEST} = R_{SRC1}$



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); result1 ←operand1;
R_{IDEST} ←Register(result1);

Description: Zero extend half

Restrictions: No address or bundle restrictions.

No latency constraints.

Exceptions: None.

Appendix A Instruction encoding

This appendix describes the ST231 instruction encoding.

A.1 Reserved bits

Any bits that are not defined are reserved. These bits must be set to 0.

A.2 Fields

Each instruction encoding is composed of a number of fields representing the operands. These are listed in *Table 93*.

Table 93. Operand fields

Operand field	Description
BCOND	Branch register containing the branch condition.
BDEST	Destination branch register for register format operations.
BDEST2	Destination branch register.
BTARG	Branch offset value from PC.
DEST	Destination general purpose register for register format operations.
NLDEST	Destination general purpose register for multiply operations ($r63 cannot be used).
IBDEST	Destination branch register for immediate format operations.
IDEST	Destination general purpose register for immediate format operations.
NLIDEST	Destination general purpose register for immediate format multiplies ($r63 cannot be used).
ISRC2	9-bit short immediate value.
IMM	23-bit value used to extend a short immediate.
SCOND	Source branch register used for select condition or carry.
SRC1	General purpose source register.
SRC2	General purpose source register.
SBRKNUM	21-bit immediate operand for sbrk

A.3 Formats

Table 94. Formats

	Stop bit		Format		Opcode							Immediate/ Dest Dest/Src2								Dest/Src2						Src1					
	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
Int3R	s		00		0	0	OPC								DEST						SRC2						SRC1					
Int3I	s		00		1	0	OPC					ISRC2									IDEST						SRC1					
Monadic	s		00		1	0	01110					OPC									IDEST						SRC1					
Cmp3R_Reg	s		00		0	1	0	OPC							DEST						SRC2						SRC1					
Cmp3R_Br	s		00		0	1	1	OPC				BDEST									SRC2						SRC1					
Cmp3I_Reg	s		00		1	1	0	OPC				ISRC2									IDEST						SRC1					
Cmp3I_Br	s		00		1	1	1	OPC				ISRC2												IBDEST			SRC1					
Imm	s		01		OPC					IMM																					
SelectR	s		01		0	OPC			SCOND						DEST						SRC2						SRC1					
SelectI	s		01		1	OPC			SCOND			ISRC2									IDEST						SRC1					
cgen	s		01		OPC				SCOND			BDEST			DEST						SRC2						SRC1					
SysOp	s		01		OPC																											
SBreak	s		01		OPC							SBRKNUM																				
System	s		10		1111				111			OPC									SRC2						SRC1					
Load	s		10		OPC				D			ISRC2									IDEST						SRC1					
Store	s		10		OPC							ISRC2									SRC2						SRC1					
Psw	s		10		OPC																SRC2						SRC1					
Call	s		11		0	OPC			LNK			BTARG																				
Branch	s		11		1	OPC	BCOND					BTARG																				
Mul64R	s		00		0	1	OPC								NLDEST						SRC2						SRC1					
Mul64I	s		00		1	1	OPC					ISRC2									NLIDEST						SRC1					
AsmR_Reg	s		10		1	1	OPC								DEST						SRC2						SRC1					
AsmI_Reg	s		10		1	1	OPC					ISRC2									IDEST						SRC1					

Important points to note.

- The **stop** bit indicates the end of bundle and is set in the last syllable of the bundle.
- The format bits are used to decode the class of operation. There are four formats:

Integer	arithmetic, comparison
Specific	immediate extension, selects, extended arithmetic
Memory	load, store
Control transfer	branch, call, rfi, goto

- Additional decoding is performed using the most significant instruction bits.
- **Int3** operations have two base formats, register (**Int3R**) and immediate (**Int3I**). Bit 27 specifies the Int3 format, 0=register format, 1=immediate format. In register format, the operation consists of R_{DEST} = R_{SRC1} **Op** RSRC2. Immediate format consists of R_{DEST} = R_{SRC1} **Op** IMMEDIATE.
- **Cmp3** format is similar to **Int3** except it can have as a destination either a general purpose register or a branch register (BBDEST). In register format, the target register specifier occupies bits 12 to 17, while the target branch register bits 18 to 20. In immediate format, bits 6 to 11 specify either the target general purpose register or target branch register (bits 6 to 8).
- **Load** operations follow R_{DEST} = Mem[R_{SRC1} + IMMEDIATE] semantics, while **store**s follow Mem[R_{SRC1} + IMMEDIATE] = R_{SRC2}. Thus bits 6 to 11 specify either the target destination register (R_{DEST}) or the second operand source register (R_{SRC2}), depending on whether the operation is a **`load`** or **store**.

A.4 Opcodes

Table 95. Instruction 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
add	s		00		0	0	00000								*DEST*						*SRC2*						*SRC1*					
sub	s		00		0	0	00001								*DEST*						*SRC2*						*SRC1*					
shl	s		00		0	0	00010								*DEST*						*SRC2*						*SRC1*					
shr	s		00		0	0	00011								*DEST*						*SRC2*						*SRC1*					
shru	s		00		0	0	00100								*DEST*						*SRC2*						*SRC1*					
sh1add	s		00		0	0	00101								*DEST*						*SRC2*						*SRC1*					
sh2add	s		00		0	0	00110								*DEST*						*SRC2*						*SRC1*					
sh3add	s		00		0	0	00111								*DEST*						*SRC2*						*SRC1*					
sh4add	s		00		0	0	01000								*DEST*						*SRC2*						*SRC1*					
and	s		00		0	0	01001								*DEST*						*SRC2*						*SRC1*					
andc	s		00		0	0	01010								*DEST*						*SRC2*						*SRC1*					
or	s		00		0	0	01011								*DEST*						*SRC2*						*SRC1*					
orc	s		00		0	0	01100								*DEST*						*SRC2*						*SRC1*					
xor	s		00		0	0	01101								*DEST*						*SRC2*						*SRC1*					
mullhus	s		00		0	0	01111								*NLDEST*						*SRC2*						*SRC1*					

Table 95. Instruction encoding (continued)

	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
max	s		00		0	0	10000								*DEST*						*SRC2*						*SRC1*					
maxu	s		00		0	0	10001								*DEST*						*SRC2*						*SRC1*					
min	s		00		0	0	10010								*DEST*						*SRC2*						*SRC1*					
minu	s		00		0	0	10011								*DEST*						*SRC2*						*SRC1*					
mulhhs	s		00		0	0	10100								*NLDEST*						*SRC2*						*SRC1*					
mull	s		00		0	0	10101								*NLDEST*						*SRC2*						*SRC1*					
mullu	s		00		0	0	10110								*NLDEST*						*SRC2*						*SRC1*					
mulh	s		00		0	0	10111								*NLDEST*						*SRC2*						*SRC1*					
mulhu	s		00		0	0	11000								*NLDEST*						*SRC2*						*SRC1*					
mulll	s		00		0	0	11001								*NLDEST*						*SRC2*						*SRC1*					
mulllu	s		00		0	0	11010								*NLDEST*						*SRC2*						*SRC1*					
mullh	s		00		0	0	11011								*NLDEST*						*SRC2*						*SRC1*					
mullhu	s		00		0	0	11100								*NLDEST*						*SRC2*						*SRC1*					
mulhh	s		00		0	0	11101								*NLDEST*						*SRC2*						*SRC1*					
mulhhu	s		00		0	0	11110								*NLDEST*						*SRC2*						*SRC1*					
mulhs	s		00		0	0	11111								*NLDEST*						*SRC2*						*SRC1*					
cmpeq	s		00		0	1	0	0000							*DEST*						*SRC2*						*SRC1*					
cmpne	s		00		0	1	0	0001							*DEST*						*SRC2*						*SRC1*					
cmpge	s		00		0	1	0	0010							*DEST*						*SRC2*						*SRC1*					
cmpgeu	s		00		0	1	0	0011							*DEST*						*SRC2*						*SRC1*					
cmpgt	s		00		0	1	0	0100							*DEST*						*SRC2*						*SRC1*					
cmpgtu	s		00		0	1	0	0101							*DEST*						*SRC2*						*SRC1*					
cmple	s		00		0	1	0	0110							*DEST*						*SRC2*						*SRC1*					
cmpleu	s		00		0	1	0	0111							*DEST*						*SRC2*						*SRC1*					
cmplt	s		00		0	1	0	1000							*DEST*						*SRC2*						*SRC1*					
cmpltu	s		00		0	1	0	1001							*DEST*						*SRC2*						*SRC1*					
andl	s		00		0	1	0	1010							*DEST*						*SRC2*						*SRC1*					
nandl	s		00		0	1	0	1011							*DEST*						*SRC2*						*SRC1*					
orl	s		00		0	1	0	1100							*DEST*						*SRC2*						*SRC1*					
norl	s		00		0	1	0	1101							*DEST*						*SRC2*						*SRC1*					
mul32	s		00		0	1	01110								*NLDEST*						*SRC2*						*SRC1*					
mul64h	s		00		0	1	01111								*NLDEST*						*SRC2*						*SRC1*					
cmpeq	s		00		0	1	1	0000				*BDEST*									*SRC2*						*SRC1*					
cmpne	s		00		0	1	1	0001				*BDEST*									*SRC2*						*SRC1*					
cmpge	s		00		0	1	1	0010				*BDEST*									*SRC2*						*SRC1*					

Table 95. Instruction encoding (continued)

	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
cmpgeu	s		00		0	1	1	0011				*BDEST*									*SRC2*						*SRC1*					
cmpgt	s		00		0	1	1	0100				*BDEST*									*SRC2*						*SRC1*					
cmpgtu	s		00		0	1	1	0101				*BDEST*									*SRC2*						*SRC1*					
cmple	s		00		0	1	1	0110				*BDEST*									*SRC2*						*SRC1*					
cmpleu	s		00		0	1	1	0111				*BDEST*									*SRC2*						*SRC1*					
cmplt	s		00		0	1	1	1000				*BDEST*									*SRC2*						*SRC1*					
cmpltu	s		00		0	1	1	1001				*BDEST*									*SRC2*						*SRC1*					
andl	s		00		0	1	1	1010				*BDEST*									*SRC2*						*SRC1*					
nandl	s		00		0	1	1	1011				*BDEST*									*SRC2*						*SRC1*					
orl	s		00		0	1	1	1100				*BDEST*									*SRC2*						*SRC1*					
norl	s		00		0	1	1	1101				*BDEST*									*SRC2*						*SRC1*					
mul64hu	s		00		0	1	11110								*NLDEST*						*SRC2*						*SRC1*					
mulfrac	s		00		0	1	11111								*NLDEST*						*SRC2*						*SRC1*					
add	s		00		1	0	00000					*ISRC2*									*IDEST*						*SRC1*					
sub	s		00		1	0	00001					*ISRC2*									*IDEST*						*SRC1*					
shl	s		00		1	0	00010					*ISRC2*									*IDEST*						*SRC1*					
shr	s		00		1	0	00011					*ISRC2*									*IDEST*						*SRC1*					
shru	s		00		1	0	00100					*ISRC2*									*IDEST*						*SRC1*					
sh1add	s		00		1	0	00101					*ISRC2*									*IDEST*						*SRC1*					
sh2add	s		00		1	0	00110					*ISRC2*									*IDEST*						*SRC1*					
sh3add	s		00		1	0	00111					*ISRC2*									*IDEST*						*SRC1*					
sh4add	s		00		1	0	01000					*ISRC2*									*IDEST*						*SRC1*					
and	s		00		1	0	01001					*ISRC2*									*IDEST*						*SRC1*					
andc	s		00		1	0	01010					*ISRC2*									*IDEST*						*SRC1*					
or	s		00		1	0	01011					*ISRC2*									*IDEST*						*SRC1*					
orc	s		00		1	0	01100					*ISRC2*									*IDEST*						*SRC1*					
xor	s		00		1	0	01101					*ISRC2*									*IDEST*						*SRC1*					
sxtb	s		00		1	0	01110					000000000									*IDEST*						*SRC1*					
sxth	s		00		1	0	01110					000000001									*IDEST*						*SRC1*					
bswap	s		00		1	0	01110					000000010									*IDEST*						*SRC1*					
zxth	s		00		1	0	01110					000000011									*IDEST*						*SRC1*					
clz	s		00		1	0	01110					000000100									*IDEST*						*SRC1*					
mullhus	s		00		1	0	01111					*ISRC2*									*NLIDEST*						*SRC1*					
max	s		00		1	0	10000					*ISRC2*									*IDEST*						*SRC1*					
maxu	s		00		1	0	10001					*ISRC2*									*IDEST*						*SRC1*					

Table 95. Instruction encoding (continued)

	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
min	s		00		1	0	10010					*ISRC2*									*IDEST*						*SRC1*					
minu	s		00		1	0	10011					*ISRC2*									*IDEST*						*SRC1*					
mulhhs	s		00		1	0	10100					*ISRC2*									*NLIDEST*						*SRC1*					
mull	s		00		1	0	10101					*ISRC2*									*NLIDEST*						*SRC1*					
mullu	s		00		1	0	10110					*ISRC2*									*NLIDEST*						*SRC1*					
mulh	s		00		1	0	10111					*ISRC2*									*NLIDEST*						*SRC1*					
mulhu	s		00		1	0	11000					*ISRC2*									*NLIDEST*						*SRC1*					
mulll	s		00		1	0	11001					*ISRC2*									*NLIDEST*						*SRC1*					
mulllu	s		00		1	0	11010					*ISRC2*									*NLIDEST*						*SRC1*					
mullh	s		00		1	0	11011					*ISRC2*									*NLIDEST*						*SRC1*					
mullhu	s		00		1	0	11100					*ISRC2*									*NLIDEST*						*SRC1*					
mulhh	s		00		1	0	11101					*ISRC2*									*NLIDEST*						*SRC1*					
mulhhu	s		00		1	0	11110					*ISRC2*									*NLIDEST*						*SRC1*					
mulhs	s		00		1	0	11111					*ISRC2*									*NLIDEST*						*SRC1*					
cmpeq	s		00		1	1	0	0000				*ISRC2*									*IDEST*						*SRC1*					
cmpne	s		00		1	1	0	0001				*ISRC2*									*IDEST*						*SRC1*					
cmpge	s		00		1	1	0	0010				*ISRC2*									*IDEST*						*SRC1*					
cmpgeu	s		00		1	1	0	0011				*ISRC2*									*IDEST*						*SRC1*					
cmpgt	s		00		1	1	0	0100				*ISRC2*									*IDEST*						*SRC1*					
cmpgtu	s		00		1	1	0	0101				*ISRC2*									*IDEST*						*SRC1*					
cmple	s		00		1	1	0	0110				*ISRC2*									*IDEST*						*SRC1*					
cmpleu	s		00		1	1	0	0111				*ISRC2*									*IDEST*						*SRC1*					
cmplt	s		00		1	1	0	1000				*ISRC2*									*IDEST*						*SRC1*					
cmpltu	s		00		1	1	0	1001				*ISRC2*									*IDEST*						*SRC1*					
andl	s		00		1	1	0	1010				*ISRC2*									*IDEST*						*SRC1*					
nandl	s		00		1	1	0	1011				*ISRC2*									*IDEST*						*SRC1*					
orl	s		00		1	1	0	1100				*ISRC2*									*IDEST*						*SRC1*					
norl	s		00		1	1	0	1101				*ISRC2*									*IDEST*						*SRC1*					
mul32	s		00		1	1	01110					*ISRC2*									*NLIDEST*						*SRC1*					
mul64h	s		00		1	1	01111					*ISRC2*									*NLIDEST*						*SRC1*					
cmpeq	s		00		1	1	1	0000				*ISRC2*												*IBDEST*			*SRC1*					
cmpne	s		00		1	1	1	0001				*ISRC2*												*IBDEST*			*SRC1*					
cmpge	s		00		1	1	1	0010				*ISRC2*												*IBDEST*			*SRC1*					
cmpgeu	s		00		1	1	1	0011				*ISRC2*												*IBDEST*			*SRC1*					
cmpgt	s		00		1	1	1	0100				*ISRC2*												*IBDEST*			*SRC1*					

Table 95. Instruction encoding (continued)

	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
cmpgtu	s		00		1	1	1	0101				*ISRC2*												*IBDEST*			*SRC1*					
cmple	s		00		1	1	1	0110				*ISRC2*												*IBDEST*			*SRC1*					
cmpleu	s		00		1	1	1	0111				*ISRC2*												*IBDEST*			*SRC1*					
cmplt	s		00		1	1	1	1000				*ISRC2*												*IBDEST*			*SRC1*					
cmpltu	s		00		1	1	1	1001				*ISRC2*												*IBDEST*			*SRC1*					
andl	s		00		1	1	1	1010				*ISRC2*												*IBDEST*			*SRC1*					
nandl	s		00		1	1	1	1011				*ISRC2*												*IBDEST*			*SRC1*					
orl	s		00		1	1	1	1100				*ISRC2*												*IBDEST*			*SRC1*					
norl	s		00		1	1	1	1101				*ISRC2*												*IBDEST*			*SRC1*					
mul64hu	s		00		1	1	11110					*ISRC2*												*NLIDEST*			*SRC1*					
mulfrac	s		00		1	1	11111					*ISRC2*												*NLIDEST*			*SRC1*					
slct	s		01		0	0	0	*SCOND*							*DEST*						*SRC2*						*SRC1*					
slctf	s		01		0	0	1	*SCOND*							*DEST*						*SRC2*						*SRC1*					
addcg	s		01		0	0	1	0	*SCOND*			*BDEST*			*DEST*						*SRC2*						*SRC1*					
divs	s		01		0	1	0	0	*SCOND*			*BDEST*			*DEST*						*SRC2*						*SRC1*					
imml	s		01		0	1	0	1	0	*IMM*																						
immr	s		01		0	1	0	1	1	*IMM*																						
slct	s		01		1	0	0	*SCOND*				*ISRC2*									*IDEST*						*SRC1*					
slctf	s		01		1	0	1	*SCOND*				*ISRC2*									*IDEST*						*SRC1*					
prgins	s		01		1	1	1	1	0	0	0																					
sbrk	1		01		1	1	1	1	0	1	1	*SBRKNUM*																				
syscall	s		01		1	1	1	1	1	1	0	*SBRKNUM*																				
break	s		01		1	1	1	1	1	1	1																					
ldw	s		10		0	0	0	0	0			*ISRC2*									*IDEST*						*SRC1*					
ldw.d	s		10		0	0	0	0	1			*ISRC2*									*IDEST*						*SRC1*					
ldh	s		10		0	0	0	1	0			*ISRC2*									*NLIDEST*						*SRC1*					
ldh.d	s		10		0	0	0	1	1			*ISRC2*									*NLIDEST*						*SRC1*					
ldhu	s		10		0	0	1	0	0			*ISRC2*									*NLIDEST*						*SRC1*					
ldhu.d	s		10		0	0	1	0	1			*ISRC2*									*NLIDEST*						*SRC1*					
ldb	s		10		0	0	1	1	0			*ISRC2*									*NLIDEST*						*SRC1*					
ldb.d	s		10		0	0	1	1	1			*ISRC2*									*NLIDEST*						*SRC1*					
ldbu	s		10		0	1	0	0	0			*ISRC2*									*NLIDEST*						*SRC1*					
ldbu.d	s		10		0	1	0	0	1			*ISRC2*									*NLIDEST*						*SRC1*					
stw	s		10		0	1	0	1	0			*ISRC2*									*SRC2*						*SRC1*					
sth	s		10		0	1	0	1	1			*ISRC2*									*SRC2*						*SRC1*					

Table 95. Instruction encoding (continued)

	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
stb	s		10		01100							*ISRC2*									*SRC2*						*SRC1*					
pft	s		10		01101							*ISRC2*									000000						*SRC1*					
prgadd	s		10		01110							*ISRC2*									000000						*SRC1*					
prgset	s		10		01111							*ISRC2*									000000						*SRC1*					
sync	s		10		10000																000000						000000					
prginspg	s		10		10001							*ISRC2*									000000						*SRC1*					
pswset	s		10		10010																*SRC2*						000000					
pswclr	s		10		10011																*SRC2*						000000					
call	s		11		0	000			0	*BTARG*																						
call	s		11		0	000			1	00000000000000000000000																						
goto	s		11		0	001			0	*BTARG*																						
goto	s		11		0	001			1	00000000000000000000000																						
rfi	s		11		0	010			0	00000000000000000000000																						
br	s		11		1	0	*BCOND*			*BTARG*																						
brf	s		11		1	1	*BCOND*			*BTARG*																						

Appendix B STBus endian behavior

The processor behaves in a different manner depending on whether the ST231 is operating in big endian or in little endian mode. *Section 17.5.2: Memory model on page 130* introduces the notation used in this appendix and defines the processor's operation in terms of a logical view of memory. This appendix describes the mapping between that logical memory and an actual physical memory attached to an STBus.

B.1 Endianness of bytes and half-words within a word based memory

The STBus views memory as being constructed from an array of 32-bit words. The notation **WMEM[i]** is used to represent 32-bit words in memory where **i** varies in the range **[0, 2^{30})**, and **MEM[s]** represents a byte indexed within **WMEM[i]**.

For a little endian memory system:

$$MEM[s] = WMEM[s/4]_{<8(s\backslash 4) \text{ FOR } 8>}$$

For a big endian memory system:

$$MEM[s] = WMEM[s/4]_{<8(3-s\backslash 4) \text{ FOR } 8>}$$

Half-word accesses are made by pairing byte accesses using the equations given above.

Considering two processors of different endianness connected to the same memory system, and representing the logical memory as seen by them as $\mathbf{MEM_{LE}[i]}$ for the little endian processor and $\mathbf{MEM_{BE}[i]}$ for the big endian processor:

$$MEM_{LE}[i] = MEM_{BE}[i \oplus 3]$$

and:

$$WMEM_{LE}[i] = WMEM_{BE}[i]$$

As an example given the word **WMEM[i]**, which stores the value 0xAABBCCDD. In either endianness the word will read the same, but when read as bytes by a little endian processor:

$MEM_{LE}[i]$ = 0xDD

$MEM_{LE}[i+1]$ = 0xCC

$MEM_{LE}[i+2]$ = 0xBB

$MEM_{LE}[i+3]$ = 0xAA

When read by a big endian processor:

$MEM_{BE}[i]$ = 0xAA

$MEM_{BE}[i+1]$ = 0xBB

$MEM_{BE}[i+2]$ = 0xCC

$MEM_{BE}[i+3]$ = 0xDD

B.2 Endianness of 64-bit accesses

The ST231 has a 64-bit STBus initiator port. The data presented to the STBus is determined differently depending upon the endianness mode. The STBus also interprets the information differently.

DMEM[i] refers to a double word in memory where i varies in the range [0, 229). When a little endian processor accesses a word address s:

$WMEM_{LE}[s] = DMEM_{LE}[s/2]_{<32(s\backslash 2) \text{ FOR } 32>}$

and for a big-endian processor:

$WMEM_{BE}[s] = WMEM_{LE}[s/2]_{<32(1-s\backslash 2) \text{ FOR } 32>}$

For example, if

WMEM[*s*]=0xaaaaaaaa

WMEM[*s*+1]=0xbbbbbbbb

then the contents of DMEM are:

$DMEM_{LE}[s]$ = 0xbbbbbbbb_aaaaaaaa

$DMEM_{BE}[s]$ = 0xaaaaaaaa_bbbbbbbb

therefore the order of words within the double word has changed.

B.3 System requirements

Systems operating purely in a single mode are straightforward. All accesses as seen by the processor are consistent and behave as would be expected for a processor of that endianness.

Issues can arise where the memory system can be observed in both little-endian and big-endian modes. A correctly implemented system behaves according to the definitions given in this document. To ensure a correct implementation, the following points must be addressed in the system.

- The STBus and all devices with 64-bit target ports must be aware of the endianness of an access.
- Size convertors must be correctly configured for the endianness of the system. The correct operation of any size convertors ensure that 32-bit target ports do not need to be aware of endianness.

If a system is NOT properly configured then the following problems may be observed.

- The peripheral registers of the ST231 may appear at the wrong address; bit 2 of the address could be inverted. This can be caused by a size convertor not being aware of endianness.
- Pairs of words may be swapped in memory.
- Words may be written to the wrong address; bit[2] could be inverted.

Glossary

Branch registers	The set of eight 1-bit registers that encode the condition for conditional branches and carry bits.
Bundle	Wide instruction of multiple operations always issued during the same cycle and executed in parallel.
Cache set	A set of a cache refers to all cache lines which may contain data at a given address. For a direct mapped cache the size of the set is 1, and for an *n*-way set associative cache the size of the set is *n*.
Commit point	The point at which the results of operations are written to the architectural state of the ST231.
Control register	One of a set of address mapped registers maintained by the hardware (or operating system or user). These registers may have side effects and may require supervisor access permissions.
Core	The core is the ST231 processor core excluding peripherals.
Dyadic operation	An operation on two operands.
General-purpose registers	The set of directly addressed fixed-point registers. ST200 contains one GR file organized as a bank of 64 32-bit registers. The compiler is responsible for explicitly scheduling data transfers among GRs.
Half-word, word, long word	Half-word relates to a 16-bit data item. Word relates to a 32-bit data item. Long word relates to a 64-bit data item.
Level-1 I-cache	Level-1 instruction cache also referred as the “closest” or “lowest” cache. Similar notations apply to the Level-1 data cache. ST200 supports multiple Level-1 data caches.
Long word	See half-word definition.
LRU	Least Recently Used. A replacement policy for caches and buffers. An LRU policy will replace the oldest entry whenever there is insufficient space for a new entry.
Main memory	This is the system-accessible memory, cached.
Misaligned	A memory access is misaligned if the access does not fit the natural alignment width of the word being accessed, and the access is illegal.
Monadic operation	An operation on one operand.
Operation	An operation is an atomic ST200 action, in general considered roughly equivalent to a typical instruction of a traditional 32-bit RISC machine.
Predication	The operation of selectively quashing an operation according to the value of a register (called predicate). One of the simplest forms of predication is a *select* operation, which is supported in ST200.
Round robin	A replacement policy for caches and buffers. A round robin policy replaces entries in turn whenever there is insufficient space for a new entry.
Set	See cache set definition.

Speculative	A speculative operation (also known as “eager”) is an operation executed prior to the resolution of the branch under which the operation would normally execute. Special attention must be paid to speculative memory **load** operations to handle the possible resulting exceptions. Speculative memory **load** operation are sometimes called “dismissible” as any exception deriving from the operation has to be ignored (“dismissed”) by the system.
ST231	The ST231 is the processor core as described in this manual including the associated peripherals. Also see “core” definition.
Superscalar	An architecture with multiple functional units in which instructions are scheduled dynamically by the hardware at run-time.
Syllable	Encoded component of a bundle that specifies one operation to be executed by the machine functional units. Syllables are composed of register and/or immediate fields and opcode specifiers. A bundle in ST200 may contain multiple syllables, each of them 32-bit wide.
VLIW	Very long instruction word: instructions (called “bundles” in ST200 terminology) potentially encode multiple, independent operations, and are fully scheduled at compile time.
Word	See half-word definition.

List of instructions

Revision history

Table 96. Document revision history

Date	Revision	Changes
07-Sep-2009	N	*Chapter 14: Debugging support (JTAG)* made the following changes: – *Table 61 on page 102* updated HOST_EVENT_ACK_PENDING – *Section 14.3.2: Default debug handler* reworded section *Command loop on page 104* and *Table 63* and in section *Default handler commands on page 104*, changed “TAPLINK_EVENT_DEFAULT event” to “value 0x7” throughout. – *Table 65 on page 109* updated event command (sent from host)
03-Mar-2009	M	Made a number of changes throughout the manual to bring this manual in line with the layout of the *ST240 core and instruction set reference manual* (ADCS 8059133). Updated *Section 7.3.5: Uncached load and stores on page 53*. Updated *Chapter 12: Interrupt controller on page 80*. Updated *Section 18.4: Macros on page 141*.
26-Jun-2008	L	Corrected minor errors throughout. Added *List of instructions on page 324*.
12-Sep-2007	K	Updated the preface to reflect the current documentation suite. No technical changes.
28-Mar-2007	J	Miscellaneous updates to: *Chapter 6: Memory translation and protection on page 31* *Chapter 5: Traps (exceptions and interrupts) on page 25* *Chapter 9: Control registers on page 67* *Chapter 14: Debugging support (JTAG) on page 98* *Chapter 17: Specification notation on page 119* *Chapter 18: Instruction set on page 137*
30-Jan-2007	I	Update into new corporate template.

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

XYZ

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com