CompactRISC™

CR16B
Programmer's Reference Manual

Literature Number: 633150-001

September 1997

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
1.0	September 1997	First release.

PREFACE

This Programmer's Reference Manual presents the programming model for the CR16B microprocessor core. The key to system programming, and a full understanding of the characteristics and limitations of the CompactRISC Toolset, is understanding the programming model.

The information contained in this manual is for reference only and is subject to change without notice.

CONTENTS

Chapter 1 INTRODUCTION

Chapter 2 PROGRAMMING MODEL

2.1 CR16B SMALL AND LARGE PROGRAMMING MODELS 2-1

2.2 COMPATIBILITY WITH CR16A 2-1

2.3 DATA TYPES 2-1

2.4 INSTRUCTION SET 2-2

2.5 REGISTER SET 2-6

2.5.1 General-Purpose Registers 2-7

2.5.2 Dedicated Address Registers 2-7

2.5.3 The Processor Status Register 2-8

2.5.4 The Configuration Register 2-10

2.5.5 Debug Registers 2-11

2.6 MEMORY ORGANIZATION 2-12

2.6.1 Data References 2-14

2.6.2 Stacks 2-14

2.7 ADDRESSING MODES 2-15

Chapter 3 EXCEPTIONS

3.1 INTRODUCTION 3-1

3.1.1 General 3-1

3.1.2 Interrupt Handling 3-2

3.1.3 Traps 3-3

3.2 DETAILED EXCEPTION PROCESSING 3-4

3.2.1 Instruction Endings 3-4

3.2.2 The Dispatch Table 3-5

3.2.3 Acknowledging an Exception 3-6

3.2.4 Exception Service Procedures 3-9

3.2.5 Returning From Exception Service Procedures 3-10

3.2.6 Priority Among Exceptions 3-10

3.2.7 Nested Interrupts 3-12

3.3 RESET 3-13

Chapter 4 ADDITIONAL TOPICS

4.1 DEBUGGING SUPPORT....4-1
4.1.1 Instruction Tracing 4-1
4.1.2 Compare-Address Match 4-3
4.1.3 Checking for Debug and Breakpoint Conditions 4-4
4.1.4 Controlling the Debug and In-System-Emulator Options 4-4
4.1.5 In-System Emulator (ISE) 4-6
4.2 INSTRUCTION EXECUTION ORDER 4-7
4.2.1 The Instruction Pipeline 4-8
4.2.2 Serializing Operations 4-9

Chapter 5 INSTRUCTION SET

5.1 INSTRUCTION DEFINITIONS....6-1
5.2 DETAILED INSTRUCTION LIST 6-3
5.3 CR16B/CR16A INSTRUCTION INCOMPATIBILITIES....6-65

Appendix A INSTRUCTION EXECUTION TIMING

Appendix B INSTRUCTION SET ENCODING

Appendix C STANDARD CALLING CONVENTIONS

Appendix D COMPARISON OF CR16A AND CR16B

INDEX

FIGURES

Figure 1-1. Code Size Comparison 1-3
Figure 1-2. RAM Size Comparison 1-4
Figure 2-1. CR16B Registers 2-6
Figure 2-2. Memory Organization 2-13
Figure 2-3. Data Representation 2-14
Figure 3-1. Dispatch Table and Jump Table 3-6
Figure 3-2. Saving PC and PSR Contents During an Exception Processing Sequence . 3-8
Figure 3-3. Transfer of Control During an Exception Acknowledge Sequence 3-8
Figure 3-4. Exception Processing Flowchart 3-11
Figure 4-1. CR16B Operating States 4-7
Figure 4-2. Memory References for Consecutive Instructions 4-9
Figure 5-1. Instruction Header Format 6-1
Figure 5-2. Instruction Example Format 6-2

TABLES

Table 3-1. Summary of Exception Processing 3-9
Table 5-1. BR/BRcond Target Addressing Methodology 6-9
Table 5-2. BAL Target Addressing Methodology 6-12
Table 5-3. CBIT/SBIT/TBIT Addressing Methodology 6-17
Table 5-4. LOAD/STOR Memory Addressing Methodology 6-30

Chapter 1
INTRODUCTION

National's CompactRISC Technology

National Semiconductor's CompactRISC architecture was created from the ground up as an alternative solution to CISC and other accumulator based architectures. CompactRISC is a RISC architecture specifically designed for embedded systems. It features the best of RISC and CISC with compact code generation, low power consumption, silicon-efficient implementations, the ability to tightly integrate on-chip acceleration, I/O and memory functions, and scalability from 8- to 64-bits.

CompactRISC implementations greatly reduce the amount of silicon required for the CPU, code memory and data memory, without significantly reducing the overall performance advantages of RISC. In addition, because any processing core is only as good as its peripheral support, several key architectural decisions were made to optimize bus structures and I/O control for embedded systems in order to improve flexibility and reduce costs.

Since its introduction, the CompactRISC architecture has firmly established itself by filling a previously unmet market gap - those embedded applications that require the performance of RISC, but cannot afford the processing and cost overheads of 32-bit RISC implementations. The 16-bit members of the CompactRISC family have been particularly popular with designers because of their optimal balance of cost and performance, plus the ability to combine a very small size core with other key on-chip functions.

CR16B - 16-bit CompactRISC Processor Core

The CR16B is a second-generation 16-bit CompactRISC processor core. It is binary compatible with its predecessor, the CompactRISC CR16A, and provides expanded options for system designers. The new implementation provides:

- Expanded linear address space of 2 Mbytes for program code and data memory
- Atomic, memory-direct bit manipulation of single bits to efficiently handle semaphores, I/O triggers, etc.
- Load and Store instructions for multiple registers
- Push and Pop instructions for multiple registers
- Addition of a Hardware Multiplier Unit for fast 16-bit multiplication
- On-chip debug features, including hardware breakpoints support
- Fully synthesizable Verilog HDL format

To ensure a seamless transition for existing CompactRISC users, the CR16B provides two programming models; one that supports the new 2 Mbyte program address space, and a smaller model that provides backward compatibility with the previous CR16A core.

The CompactRISC Architecture

In many ways, the CompactRISC technology is a traditional RISC load/store processor architecture, but enhanced for embedded control functions. For example:

- The CR16B executes an optimized instruction set with 24 internal registers grouped in 16 general-purpose registers, three special function registers, a processor status register, a configuration register and three debug-control registers.
- The CR16B has a three-stage pipeline that is used to obtain a peak performance of 50 Million Instructions Per Second (MIPS) at a clock frequency of 50 MHz.
- The CR16B core includes a pipelined integer unit that supports a peak execution speed of one instruction per each internal cycle, with a 100 Mbyte/sec. pipelined bus.
- The CR16B performs fast multiply operations using a 16-bit by 4-bit hardware multiplier.

In general, the CompactRISC architecture supports little-endian memory addressing. Which means that the byte order in the CR16B is from the least significant byte to the most significant byte.

Reduced Memory Requirements

To simplify instruction decoder design, RISC architectures have traditionally employed fixed-width instructions. For 32-bit RISC systems, every instruction is encoded in four or eight bytes. In CISC systems, a variable instruction length is used, resulting in smaller code sizes for a given application. The CompactRISC architecture utilizes variable instruction widths with fixed coding fields within the instruction itself. For example, the opcode field is always in the first 16 bits with additional bytes as required for immediate values. Instructions for the CR16B may be encoded in two bytes or four bytes, but basic instructions are only two bytes long. This permits optimized instruction processing by the instruction decoder, and results in a smaller code size. Code generated for the CR16B is comparable to CISC code size, or typically 50 percent smaller than code generated for leading 32-bit RISC CPUs. Another advantage this provides, is the ability to generate performance with lower pin-counts or lower-bandwidth buses - again a trait of an embedded system.

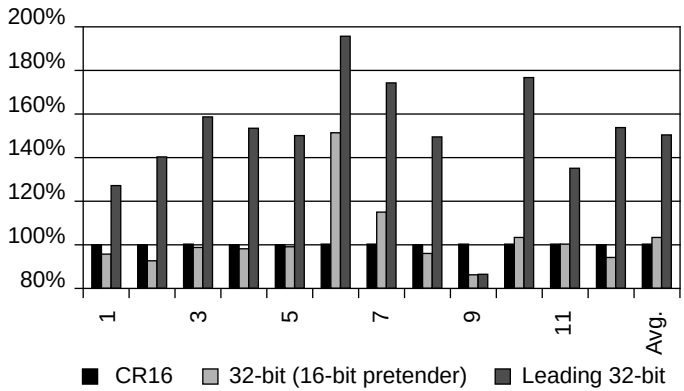


Figure 1-1. Code Size Comparison

32-bit RISC processors store registers and addresses on the stack as 32-bit entities. The CR16B is a 16-bit processor, thus it uses 16 bits for register image storage and for address storage in main memory. In addition, 32-bit RISC processors deliver high performance only when aligned 32-bit data is used. Non-aligned data significantly hampers performance. Intermediate results are stored in memory as 32-bit values and registers are saved as 32-bit operands on the stack. CompactRISC instructions operate on 8-, 16- and 32-bit data. Non-aligned accesses are allowed. Dedicated data type conversion instructions speed data access to mixed size data. With smaller code size, and variable length instructions and data, the CompactRISC family provides more efficient use of smaller, lower cost, lower bandwidth memories.

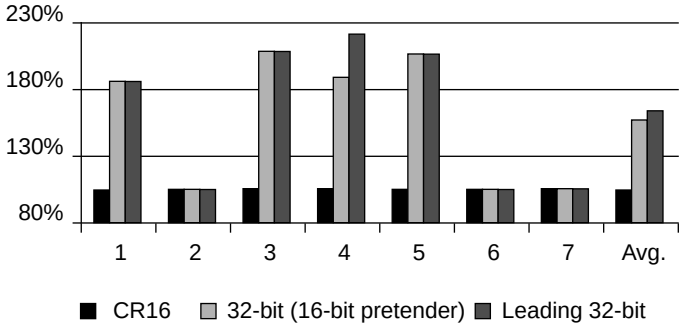


Figure 1-2. RAM Size Comparison

Smaller memory enables the designer to choose between several potential advantages:

- Reduce costs
- Integrate many more system elements with on-chip memory
- Use less pins to access minimum sized off-chip memory
- Allow larger amounts of on-chip memory than similar processors, at the same cost.

Scalable Architecture from 8 to 64 Bits

The architectural features described above make the CompactRISC technology ideal for the next generation of embedded systems. But, one additional design decision opened the door for CompactRISC technology to be effectively used from low-end to high-end embedded systems. That decision was to make the CompactRISC architecture flexible enough to accommodate the whole range of 8-bit to 64-bit implementations, thus providing a more attractive upgrade path for designers of new low-end embedded systems.

Thus, the designer of embedded controller-based systems can choose the optimum processor size for a given target application. This is particularly useful in leveraging the development investment across several classes of related end-products. With a single-processor architecture, a number of different products can be developed using a single development platform and using the same HLL-based development and debug tools. Additionally, a design team that is already experienced with a design around one CompactRISC core can easily migrate to another core due to the high similarity in both the architecture and the development tools.

Modular Extensions

The CompactRISC technology was designed to be easily extensible. This means that specialized functions needed by specific applications can be easily added to a single-chip design. A modular internal bus provides predefined processor and I/O interfaces to the core bus and the peripheral bus. These buses are designed for maximum flexibility. The core bus is a high-speed bus and can be used to connect performance-demanding functions to the CPU such as fast on-chip memory, DMA channels, and additional coprocessor units such as DSP. The peripheral bus is a simple, lower speed bus for less demanding peripherals such as counters, timers, PWM lines and MICROWIRE serial interfaces. Using a "template" approach, it is easy to create small, cost-effective custom systems. It is also easy to expand the functionality of CompactRISC core-based systems to include any number of application specific features.

Development Tools

High-level development tools are essential to rapid, modern design. The CompactRISC architecture is well supported with a comprehensive C-based development and debug environment available from National and third party vendors. Key software development components include an optimizing C compiler, a macro assembler, run-time libraries, librarian and a graphical source-level debugger including enhanced simulation capabilities. In addition, support for debugging of multiple CompactRISC/DSP cores on a single die comes from the use of an integrated, multiple core, graphical debugger. On the hardware side, the CompactRISC architecture has modular ISE (In System Emulator) support from third-party development system vendors, and various development boards for all current product offerings. As a package, these tools simplify the task of designing and developing advanced embedded systems in high level languages such as ANSI-C.

Chapter 2
PROGRAMMING MODEL

2.1 CR16B SMALL AND LARGE PROGRAMMING MODELS

The CR16B has two programming modes: small and large. The small model is similar to the CR16A, and is limited to 128 Kbytes of program address space, and 256 Kbytes of data address space. The large model supports up to 2 Mbytes of program and data space.

The two programming models are almost identical. They differ only in the instructions used for branching and program flow control. In the "INSTRUCTION SET" on page 2-2, instructions which apply only to the large model (2 Mbytes) are marked with an [L], in the description field, and instructions which apply only to the small model (128 Kbytes) are marked with an [S].

This chapter describes both programming models. Any explanation which does not specify the type of programming model, applies to both the small and the large programming models.

2.2 COMPATIBILITY WITH CR16A

The CR16B is backward-compatible with the CR16A. In other words, code that was developed for the CR16A, runs on the CR16B with almost no modifications. Appendix D provides a summary of the differences between the CR16A and the CR16B programming models.

2.3 DATA TYPES

The CompactRISC family of processors are little-endian machines. As such, the least significant bytes always resides in the lower addresses, both for address and data variables.

Integer data type

The integer data type is used to represent integers. Integers may be signed or unsigned. Two integer sizes are supported: 8-bit (1 byte), 16-bit (1 word). Signed integers are represented as binary two's complement numbers, and have values in the range -2^7 to 2^7-1 and -2^{15} to $2^{15}-1$, respectively. Unsigned numbers have values in the range 0 to 2^8-1, and 0 to $2^{16}-1$, respectively.

Boolean data type The boolean data type is represented as an integer (byte word). The value of its least significant bit represents one of two logical values, true or false. Integer 1 indicates true; integer 0 indicates false.

2.4 INSTRUCTION SET

This section includes a summary list of all the instructions in the CR16B instruction set. Appendix B, “INSTRUCTION SET ENCODING” describes each instruction in detail.

The following table summarizes the CR16B instruction set.

Mnemonic	Operands	Description
MOVES		
MOVi	Rsrc/imm, Rdest	Move
MOVX	Rsrc, Rdest	Move with sign extension
MOVZ	Rsrc, Rdest	Move with zero extension
MOVD	imm, (Rdest+1, Rdest)	Move 21-bit immediate to register-pair
INTEGER ARITHMETIC		
ADD[U]i	Rsrc/imm, Rdest	Add
ADDCi	Rsrc/imm, Rdest	Add with carry
MULi	Rsrc/imm, Rdest	Multiply: Rdest(8):= Rdest(8) * Rsrc(8)/Imm Rdest(16):= Rdest(16) * Rsrc(16)/Imm
MULSB	Rsrc, Rdest	Multiply: Rdest(16):= Rdest(8) * Rsrc(8)
MULSW	Rsrc, Rdest	Multiply: (Rdest+1, Rdest):= Rdest(16) * Rsrc(16)
MULUW	Rsrc, Rdest	Multiply: Rsrc = {R0,R1,R8,R9 only} (Rdest+1,Rdest):= Rdest(16) * Rsrc(16);
SUBi	Rsrc/imm, Rdest	Subtract: (Rdest := Rdest − Rsrc)
SUBCi	Rsrc/imm, Rdest	Subtract with carry: (Rdest := Rdest − Rsrc)
INTEGER COMPARISON		
CMPi	Rsrc/imm, Rdest	Compare (Rdest − Rsrc)
BEQ0i	Rsrc, disp	Compare Rsrc to 0 and branch if EQUAL Rsrc = (R0,R1,R8,R9 only)
BNE0i	Rsrc, disp	Compare Rsrc to 0 and branch if NOT-EQUAL Rsrc = (R0,R1,R8,R9 only)
BEQ1i	Rsrc, disp	Compare Rsrc to 1 and branch if EQUAL Rsrc = (R0,R1,R8,R9 only)
BNE1i	Rsrc, disp	Compare Rsrc to 1 and branch if NOT-EQUAL Rsrc = (R0,R1,R8,R9 only)

Mnemonic	Operands	Description
LOGICAL AND BOOLEAN		
ANDi	Rsrc/imm, Rdest	Logical AND
ORi	Rsrc/imm, Rdest	Logical OR
Scond	Rdest	Save condition code as boolean
XORi	Rsrc/imm, Rdest	Logical exclusive OR
SHIFTS		
ASHUi	Rsrc/imm, Rdest	Arithmetic left/right shift
LSHi	Rsrc/imm, Rdest	Logical left/right shift
BITS		
TBIT	Rposition/imm, Rsrc	Test bit in register
SBITi	Iposition, 0(Rbase) Iposition, disp16(Rbase) Iposition, abs	Set a bit in memory; Rbase = (R0, R1, R8, R9}
CBITi	Iposition, 0(Rbase) Iposition, disp16(Rbase) Iposition, abs	Clear a bit in memory Rbase = (R0, R1, R8, R9}
TBITi	Iposition, 0(Rbase) Iposition, disp16(Rbase) Iposition, abs	Test a bit in memory Rbase = (R0, R1, R8, R9}
JUMPS AND LINKAGE		
Bcond	disp9 disp17 disp21	Conditional branch using a 9-bit displacement Conditional branch to a small address [S] Conditional branch to a large address [L]
BAL	Rlink, disp17 (Rlink+1, Rlink), disp21	Branch and link to a small address [S] Branch and link to a large address [L]
BR	disp9 disp17 disp21	Branch using a 9-bit displacement Branch to a small address [S] Branch to a large address [L]
EXCP	vector	Trap (vector)
Jcond	Rtarget (Rtarget+1, Rtarget)	Conditional Jump to a small address [S] Conditional Jump to a large address [L]
JAL	Rlink, Rtarget (Rlink+1, Rlink), (Rtarget+1, Rtarget)	Jump and link to a small address [S] Jump and link to a large address [L]
JUMP	Rtarget (Rtarget+1, Rtarget)	Jump to a small address [S] Jump to a large address [L]
RETX		Return from exception
PUSH	imm, Rsrc	Push “imm” number of registers on user stack, starting with Rsrc
POP	imm, Rdest	Restore “imm” number of registers from user stack, starting with Rdest

Mnemonic	Operands	Description
POPRET	imm, Rdest	Restore registers (similar to POP) and perform JUMP RA or JUMP (RA, ERA), depending on memory model

PROCESSOR REGISTER MANIPULATION

LPR	Rsrc, Rproc	Load processor register
SPR	Rproc, Rdest	Store processor register

LOAD AND STORE

LOADi	disp(Rbase), Rdest	Load (register relative)
	abs, Rdest	Load (absolute)
	disp(Rpair+1, Rpair), Rdest	Load (far-relative)
STORi	Rsrc, disp(Rbase)	Store (register relative)
	Rsrc, disp(Rpair +1, Rpair)	Store (far-relative)
	Rsrc, abs	Store (absolute)
	sm_imm, 0(Rbase)	Store small immediate in memory;
	sm_imm, disp(Rbase)	Rbase = (R0, R1, R8, R9)
	sm_imm, abs	
LOADM	imm	Load 1 to 4 registers (R2 - R5) from memory, starting at the address in R0, according to imm count value
STORM	imm	Store 1 to 4 registers (R2 - R5) to memory, starting at the address in R1, according to imm count value

MISCELLANEOUS

DI		Disable maskable interrupts
EI		Enable maskable interrupts
NOP		No operation
WAIT		Wait for interrupt
EIWAIT		Enable interrupts and wait for interrupt

Instructions Table Glossary	
Rsrc	- Source Register. Any general-purpose register (R0 - R13, ERA, RA, SP)
Rsrcr	- A restricted subset of the general-purpose register set (R0, R1, R8, R9)
Rdest	- Destination Register. Any general-purpose register (R0 - R13, ERA, RA, SP)
Rproc	- Processor registers (PC, PSR, ISP, INTBASEH/L)
Rlink	- Link Register. Any general-purpose register (R0 - R13, ERA, RA, SP), holding the address of the next sequential instruction, used as a return address.
imm	- Immediate value
sm_imm	- Small immediate - 4-bit data
Rbase	- Base register. Any general-purpose register (R0 - R13, ERA, RA, SP), holding the base address of a memory variable
disp(Rbase)	- Relative addressing; Address = disp + (Rbase)
Rpair	- Any pair of sequential general-purpose registers (R0 - R13, ERA, RA)
Rtarget	- Target Register. Any general-purpose register (R0 - R13, ERA, RA, SP), holding the JUMP/JAL target address
Rposition	- Bit position, stored in any general-purpose register (R0 - R13, ERA, RA, SP).
Iposition	- Bit position, specified as an immediate operand
Rsrc(8)	- The LSB(8-bits) of any Source Register (R0 - R13, ERA, RA, SP)
Rdest(8)	- The LSB(8-bits) of any Destination Register (R0 - R13, ERA, RA, SP)
Rsrc(16)	- The complete 16-bits of any Source Register (R0 - R13, ERA, RA, SP)
Rdest(16)	- The complete 16-bits of any Destination Register (R0 - R13, ERA, RA, SP)
abs	- Absolute address
small address	- Address in the range of 0 to 128 Kbytes
large address	- Address in the range of 0 to 2 Mbytes
[L]	- Instructions exclusive to the large memory model
[S]	- Instructions exclusive to the small memory model
disp16	- 16-bit displacement

2.5 REGISTER SET

This section describes each register, its bits and its fields in detail. In addition, the format of each register is illustrated.

All registers are 16 bits wide, except for the four address registers, which are 21 bits wide. Bits specified as “reserved” must be written as 0, and return undefined results when read.

The internal registers of the CR16B are grouped by function:

- 16 general-purpose registers
- Eight processor registers:
  - Three dedicated address registers
  - One processor status register
  - One configuration register
  - Three debug control registers

Figure 2-1 shows the internal registers of the CR16B

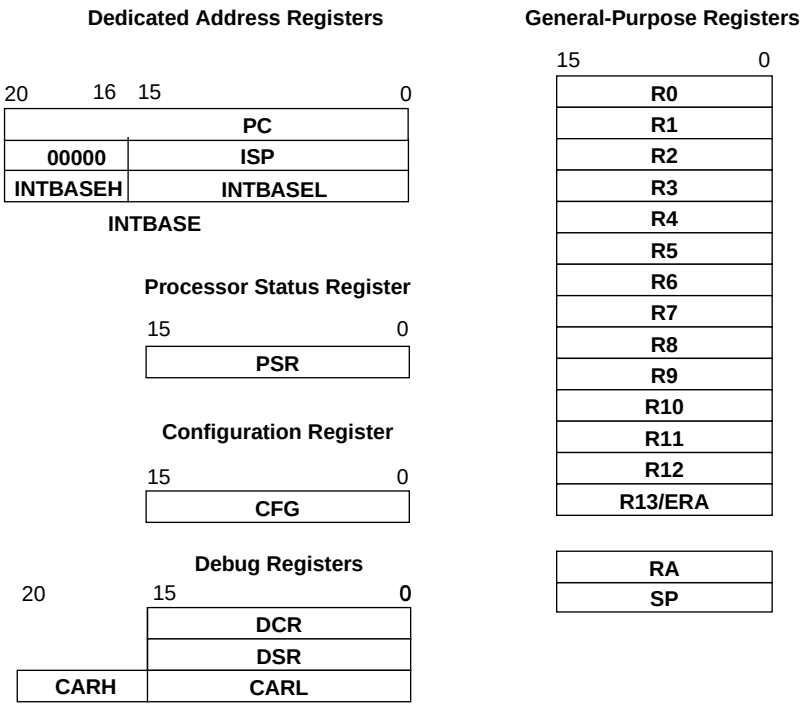


Figure 2-1. CR16B Registers

2.5.1 General-Purpose Registers

The CompactRISC cores feature 16 general purpose registers. Due to the subroutine calling conventions advocated for the architecture, some of these registers are assigned special, hardware and software functions.

Small programming model

Registers R0 - R13 are used for general purposes, such as holding variables, addresses or index values. The SP general-purpose register is usually used as a pointer to the program run-time stack. The RA general-purpose register is usually used as a return address from sub-routines. If a general-purpose register is specified by an operation that is 8 bits long, then only the low part of the register is used; the high part is not referenced, or modified.

Large programming model

Registers R0 - R12 are used for general purposes, such as holding variables, addresses or index values. The SP general-purpose register is usually used as a pointer to the program run-time stack. The (RA, ERA) general-purpose register-pair, is usually used as a return address from sub-routines, with RA containing the high address bits (PC(20:17)), and ERA containing the low address bits (PC(16:1)). If a general-purpose register is specified by an operation that is 8 bits long, only the low part of the register is used; the high part is not referenced or modified.

Notes

1. The **POPRET** instruction assumes that the return address is placed in the RA (small model) or [RA, ERA] register-pair (large model) during register restore. Thus, a program that does not use this calling convention, can not use this instruction.

2. For the Large Programming Model, only the ERA mnemonics should be used, and not R13.

3. Some of the instructions can use only a sub-set of the general-purpose registers. These appear in “INSTRUCTION SET” on page 2-2, with an “R” in the first column, and in Appendix B.

2.5.2 Dedicated Address Registers

This section describes the three, 21-bit wide, dedicated address registers that the CR16B uses to implement specific address functions.

Program Counter (PC)

The value in the PC register points to the first byte of the instruction currently being executed. The least significant bit of the PC is always 0. Thus CR16B instructions are aligned to even addresses. At reset, the PC is initialized to 0, and the values of the PC, prior to reset, are saved in the R0 general-purpose register, and the values of bits 17 through 20 are saved in the most significant bits of R1. In the small programming model, bits 17 through 20 are always 0. Therefore, in the large programming model, the instructions reside in the range 0 to $1FFFFE_{16}$, while in the small model, the instructions reside in the range 0 to $1FFFE_{16}$. The lower 12 bits of R1 contain the lower 12 bits of PSR prior to reset.

Interrupt Stack Pointer (ISP)

The ISP register points to the lowest address of the last item stored on the interrupt stack. This stack is used by the hardware when interrupts or exceptions occur. The five most significant bits, and the least significant bit of this register, are always 0. Thus the interrupt stack always starts at an even address, and resides in the address range 0 to $FFFE_{16}$. The ISP cannot be used for any purpose other than the automatic storage of registers on the stack during an exception, and the restoration of these registers during a RETX.

Interrupt Base Register (INTBASE)

The INTBASE register holds the address of the dispatch table for interrupts and traps. This register is derived from the concatenation of bits 0 through 15 of INTBASEL and bits 0 through 4 of INTBASEH, forming together a 21-bit address. Bit 0 of INTBASEL and bits 5 through 15 of INTBASEH are always 0. INTBASEH is used for dispatch-table entry calculation only when working with 21-bit entry dispatch tables (i.e., CFG.ED = 1). Otherwise, INTBASEH is considered as 0, when calculating INTBASE. Thus, in this model, the dispatch table always resides in the address range 0 to $0FFFE_{16}$. or $1FFFFE_{16}$, according to CFG.ED. See Chapter 3, “EXCEPTIONS” for more information.

2.5.3 The Processor Status Register

The Processor Status Register (PSR) holds status information and selects operating modes for the CR16B. It is 16 bits wide.

The format of the PSR is shown below:

15 12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	I	P	E	0	N	Z	F	0	0	L	T	C

At reset, bits 0 through 11 of the PSR are cleared to 0, except for the PSR.E bit, which is set to 1. In addition, the value of each bit prior to reset is saved in the R1 general-purpose register.

Several bits in the PSR have a dedicated condition code in conditional branch instructions. These bits are Z, C, L, N, and F. Any conditional branch instruction can cause a branch in the program execution, based on the value of one of these PSR bits, or a combination of them. For example, one of the **Bcond** instructions, **BEQ** (Branch EQual), causes a branch if the PSR.Z flag is set. Refer to the **Bcond** instruction in "Instruction Definitions" on page 5-1 for details.

Bits 3, 4 and 8 have a constant value of 0. Bits 12 through 15 of the PSR register are reserved. The other bits are described below. In general, status bits are modified only by instructions which are specified to do so. Otherwise, status bits maintain their values, throughout instructions which do not implicitly affect them.

The C Bit The Carry bit indicates whether a carry or borrow occurred after addition or subtraction. It can be used with the **ADDC** and **SUBC** instructions to perform multiple-precision integer arithmetic calculations. It is cleared to 0 if no carry or borrow occurred, and set to 1 if a carry or borrow occurred.

The T Bit The Trace bit causes program tracing. While the T bit is set to 1, a Trace (TRC) trap is executed after every instruction. Refer to "Instruction Tracing" on page 4-1 for more information on program tracing. The T bit is automatically cleared to 0, when a trap or an interrupt occurs. The T bit is used in conjunction with the P bit, see below.

The L Bit The Low flag is set by comparison operations. In integer comparison, the L flag is set to 1, if the second operand (*Rdest*) is less than the first operand (*Rsrc*) when both operands are interpreted as unsigned integers. Otherwise, it is cleared to 0.
Refer to the specific compare instruction in "Instruction Definitions" on page 5-1 for details.

The F Bit The Flag bit is a general condition flag which is set by various instructions. It may be used to signal exceptional conditions or to distinguish the results of an instruction (e.g., integer arithmetic instructions use it to indicate overflow from addition or subtraction). In addition, it is set, or cleared, as a result of a Test-Bit, Set-Bit or Clear-Bit instruction.

The Z Bit The Zero bit is set by comparison operations. In integer comparisons it is set to 1 if the two operands are equal. Otherwise, it is cleared to 0. Refer to the specific compare instruction in "Instruction Definitions" on page 5-1 for details.

The N Bit The Negative bit is set by comparison operations. In integer comparison it is set to 1 if the second operand (*Rdest*) is less than the first operand (*Rsrc*) when both operands are interpreted as signed integers. Otherwise it is cleared to 0.
Refer to the specific compare instruction in “Instruction Definitions” on page 5-1 for details.

The E Bit The local maskable interrupt Enable bit affects the state of maskable interrupts. While this bit and the PSR.I bits are 1, all maskable interrupts are accepted. While this bit is 0, only the non-maskable interrupt is accepted.
See “Interrupt Handling” on page 3-2.

There are two dedicated instructions that set and clear the E bit. It is set to 1 by the Enable Interrupts instruction (**EI**). It is cleared to 0 by the Disable Interrupts instruction (**DI**). This pair can be used to locally disable maskable interrupts, regardless of the global state of maskable interrupts, which is determined by the value of the PSR.I bit.
See also “Interrupt Handling” on page 3-2.

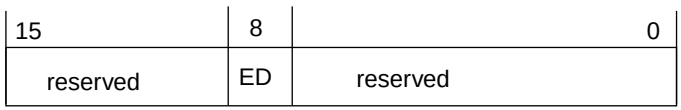
The P Bit The Trace (TRC) trap Pending bit is used together with the T bit to prevent a TRC trap from occurring more than once for any instruction. It may be cleared to 0 (no TRC trap pending) or 1 (TRC trap pending).
See “Exception Service Procedures” on page 3-9 and “Instruction Tracing” on page 4-1 for more information.

The I Bit The global maskable Interrupt enable bit affects the state of maskable interrupts. While this bit, and the PSR.E bits, are 1, all maskable interrupts are accepted. While this bit is 0, only the non-maskable interrupt is accepted. The I bit is cleared to 0 on reset. In addition, it is automatically cleared when an interrupt or DBG trap occurs.

2.5.4 The Configuration Register

The ED Bit The ED bit determines the size of INTBASE register and the size of every entry in the Interrupt Dispatch Table. When ED is set, INTBASE is a 21-bit register (INTBASEH, INTBASEL) and the Interrupt Dispatch Table contains 21-bits elements, each occupying two adjacent words. When ED is cleared, INTBASE is a 16-bit register, and the Interrupt Dispatch Table contains 16-bit elements. On reset the bit is cleared (CR16A compatible mode).

The format of the CFG register is shown below.



2.5.5 Debug Registers

The three dedicated registers, the Debug Control Register (DCR), the Debug Status Register (DSR), the Compare-Address Register (CAR), control and monitor a major portion of the debugging support available in the CR16B. The functionality of these registers, and their usage, is explained in detail in “DEBUGGING SUPPORT” on page 4-1.

The Debug Status Register (DSR)

The Debug Status Register (DSR) indicates debug and breakpoint conditions that have been detected. When the CPU detects an enabled debug condition, it sets the appropriate bits in the DSR to 1. For further information, see “Checking for Debug and Breakpoint Conditions” on page 4-4.

The Debug Control Register (DCR)

The Debug Control Register (DCR) controls the compare-address match, external tag on fetch, and PC match debug, options. These options are enabled and controlled by the appropriate bits in the DCR. For more information, see “Controlling the Debug and In-System-Emulator Options” on page 4-4.

The Compare Address Register (CAR)

The Compare Address Registers Low and High (CARL, CARH) contain the address to be used for generating a breakpoint by the on-core breakpoint unit, in case of PC match or data transactions address match. For more information, see “PC match” on page 4-2 and “Compare-Address Match” on page 4-3.

2.6 MEMORY ORGANIZATION

The CR16B implements 21-bit addresses. This allows the CPU to access up to 256 Kbytes of data, and 128 Kbytes of program memory in the small model, and 2 Mbytes of program and data in the large model. The memory is a uniform linear address space. Memory locations are numbered sequentially starting at 0 and ending at $2^{18}-1$ in the small model, and at $2^{21}-1$ in the large model. The number specifying a memory location is called an address.

CR16B data addressing is always byte-related (i.e., data can be addressed at byte-resolution). The instructions, by contrast, are always word-aligned, and therefore instruction addresses are always even-addressed.

Memory can be logically divided into the following regions: (see Figure 2-2).

- 0-63K (0 through $0FBFF_{16}$)
  This region can be accessed efficiently for data-manipulation, using register-relative and absolute addressing. Therefore, it should be used for RAMs and frequently accessed I/O devices. Also, the user-stack and interrupt-stack must be located in this region, because the SP and ISP registers are only 16-bit wide. There are no restrictions on code in this region.

- 63K-64K ($0FC00_{16}$ through $0FFFF_{16}$)
  This region is reserved for I/O devices, such as the Interrupt Controller Unit and its acknowledge address, as well as other internal uses.

- 64K - 256K (10000_{16} through 20000_{16})
  This region can be accessed efficiently using far-relative and absolute addressing. Register-relative addressing (near-pointers) has only limited addressing capabilities in this region.
  For more information, see Table 5-4, “LOAD/STOR Memory Addressing Methodology” on page 5-30, and “Medium displacement values” on page B-5.
  Use this region for infrequently used data variables, or for code. Note that only in the large memory model is the 128K - 256K also available for code.

- 256K - 2M (20000_{16} through $1FFFFF_{16}$)
  This region can only be accessed by the far-relative mode, and in a limited manner. Use this zone only in the large memory model, for infrequently used data, and for code.
  For more information, see Table 5-4, “LOAD/STOR Memory Addressing Methodology” on page 5-30, and “Far- Relative Addressing Mode” on page B-5.

For more information, see “Detailed Instruction List” on page 5-3, “Addressing Modes” on page 2-15, and “CR16B Instruction Set Summary” on page B-12.

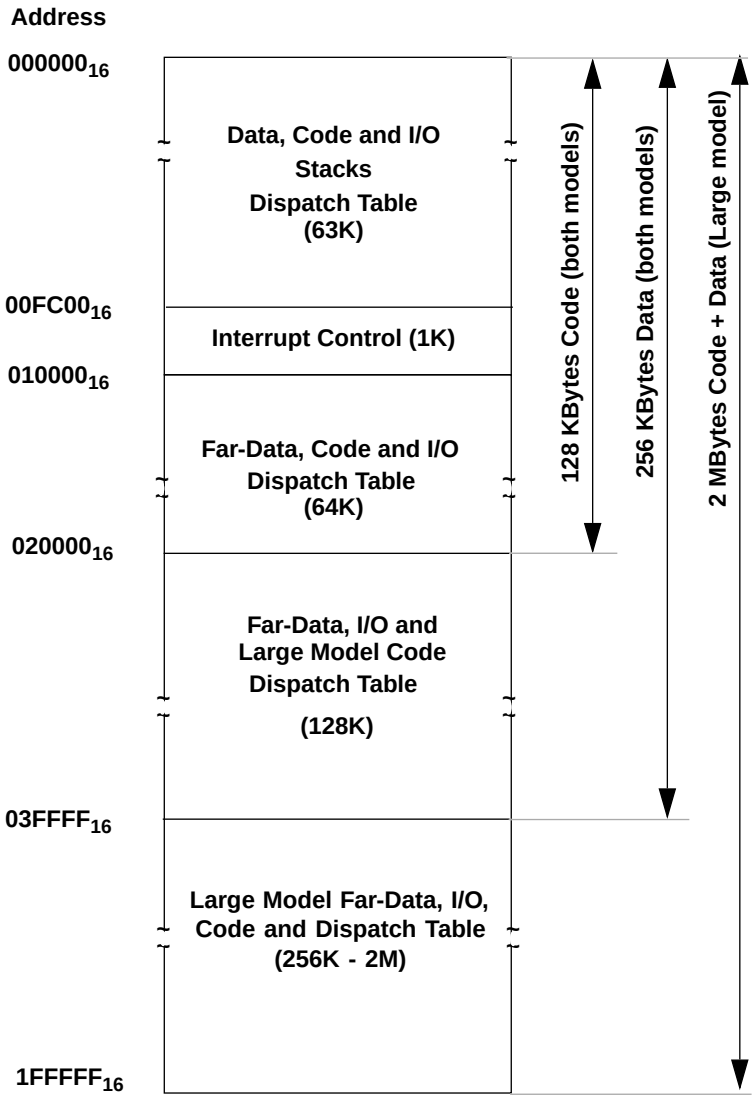


Figure 2-2. Memory Organization

2.6.1 Data References

Bit and byte order for data references

Data memory is byte-addressable in the CompactRISC architecture, and is organized in "little-endian" format, where the least significant byte within a word or a double-word resides at the lower address.

Bits are ordered from least significant to most significant. The least-significant bit is in position zero. The **TBIT**, **SBIT**, and **CBIT** instructions refer to bits by their ordinal position numbers. Figure 2-3 shows the memory representation for data values.

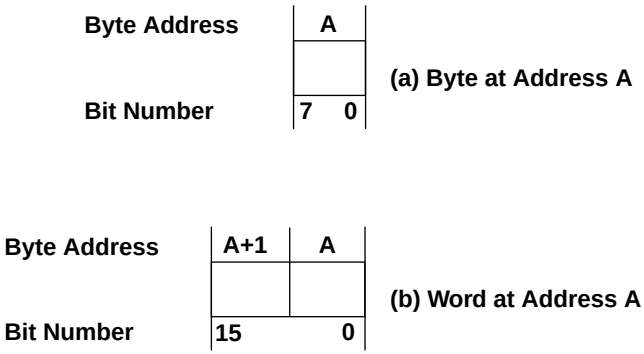


Figure 2-3. Data Representation

Data

The CR16B supports references to memory by the load and store instructions, as well as **TBIT**, **SBIT**, **CBIT**, **PUSH,POP,LOADM** and **STORM**. Bytes, and words can be referenced on any boundary.

2.6.2 Stacks

A *stack* is a one-dimensional data structure. Values are entered and removed, one item at a time, at one end of the stack called the *top-of-stack*. The stack consists of a block of memory, and a variable called the *stack pointer*. Stacks are important data structures in both systems and applications programming. They are used to store status information during sub-routine calls and interrupt servicing. In addition, algorithms for expression evaluation in compilers and interpreters use stacks to store intermediate results. High level languages, such as C, keep local data and other information on a stack.

The CR16B supports two kinds of stacks: the interrupt stack and the program stack.

The interrupt stack

The processor uses the interrupt stack to save and restore the program state during the handling of an exception condition. This information is automatically pushed, by the hardware, on to the interrupt stack before entering an exception service procedure. On exit from the exception service procedure, the hardware pops this information from the interrupt stack. See Chapter 3, "EXCEPTIONS" for more information. The interrupt stack can reside in the first 64 Kbytes of the address range, and is accessed via the ISP processor register.

The program stack

The program stack is normally used by programs at run time, to save and restore register values upon procedure entry and exit. It is also used to store local and temporary variables. The program stack is accessed via the SP general-purpose register and therefore must reside in the first 64 Kbytes of the address range. Note that this stack is handled by software only, e.g., the CompactRISC C Compiler generates code that pushes data on to, and pops data from, the program stack. Only **PUSH** and **POP** instructions adjust the **SP** automatically; otherwise, software must manage the **SP** during save and restore operations.

Both stacks expand downward in memory, toward address zero.

2.7 ADDRESSING MODES

The CompactRISC is a load/store architecture, with most instructions operating only on registers.

Most instructions use one, or two, of the CR16B registers (or register pairs) as operands. Some instructions may also use an immediate value instead of the first register operand.

Memory is accessed only by the generic Load and Store instructions, which use the absolute relative, addressing mode, or by a more function-specific set of instructions. These are the memory-direct bit manipulation instructions (**SBIT**, **CBIT**, **TBIT**), which use absolute and relative addressing, the **PUSH/POP** instructions, which can only access the stack via **SP**, and the **LOADM** and **STORM** instructions, which use a fixed set of registers for addressing.

The following addressing modes are available:

Register mode

In register mode, the operand is located in a general-purpose register, i.e., R0 through R13, RA or SP. The following instruction illustrates register addressing mode.

```
ADDB R1, R2
```

Some CR16B instructions use a register-pair to form, or store, the address,

```
JAL (RA,ERA), (R9,R8)
```

where the RA, ERA register-pair is concatenated to form a 20-bit return address, and R9 and R8 are concatenated to form a 20-bit target address.

Immediate mode

In immediate mode, the operand is a constant value which is specified within the instruction. For example:

```
MULW $4, R4
```

PC-Relative mode

In PC-Relative mode, the operand is a displacement relative to the current value of the PC register. For example:

```
BR *+10
```

Relative mode

In relative mode, the operand is located in memory. Its address is obtained by adding the contents of a general-purpose register to the constant value in the displacement field encoded in the instruction.

Relative mode limits accesses to the first 256 Kbytes of memory (18-bit displacement), but only the first 64 Kbytes may be covered with the register as a base-pointer since the register used has only 16 bits.

The following instruction illustrates relative addressing mode.

```
LOADW 12(R5), R6
```

Restricted relative mode

In some cases (**SBITi**, **CBITi**, **TBITi**, **STORi $imm**, **<>**), the displacement is limited to 16 bits, and only four registers (**R0**, **R1**, **R8**, **R9**) can be used in the addressing. In these cases, addressing is limited to the first 64 Kbytes of memory.

The following instruction illustrates relative addressing mode.

```
SBITW $3, 12(R0)
```

Far-relative mode

In far-relative mode, the operand is located in memory. Its address is obtained by concatenating a pair of adjacent general-purpose registers to form a 21-bit value, and adding this value to the 18-bit signed constant value in the displacement field encoded in the instruction. The 16 least significant bits of the 21-bit value are taken from the base register, and the five most significant bits of the value are taken from the least significant bits in the next consecutive register.

Utilizing the 18-bit signed displacement, this addressing mode can index a 2-Mbyte segment of memory, which begins anywhere within the first 128 Kbytes of memory. Only the **LOAD** and **STOR** instructions feature this mode, and can access beyond the 64K boundary. The following instruction illustrates far-relative addressing mode.

```
STORW R7, 4(R3, R2)
```

Absolute mode In absolute mode, the operand is located in memory, and its address is specified within the instruction (18 bit). The following example illustrates absolute addressing mode.

```
LOADB 4000, R6
```

For a more detailed explanation, see “Instruction Formats” on page B-2.

Chapter 3
EXCEPTIONS

3.1 INTRODUCTION

3.1.1 General

Program *exceptions* are conditions that alter the normal sequence of instruction execution, causing the processor to suspend the current process, and execute a special service procedure, often called a handler.

Exception types

An exception resulting from the activity of a source external to the processor is known as an *interrupt*; an exception which is initiated by some action or condition in the program itself is called a *trap*. Thus, an interrupt need have no relationship to the executing program, while a trap is caused by the executing program and recurs each time the program is executed. The CR16B recognizes ten exceptions: seven traps and three types of interrupts.

The exception-handling technique employed by an interrupt-driven processor determines how fast the processor can perform input/output transfers, the speed for transfers between tasks and processes, and the software overhead required for both these activities. Thus, to a large extent, it determines the efficiency of a processor's multiprogramming and multi-tasking (including real-time) capabilities.

Addressing exceptions

Exception handling in the CR16B uses a Dispatch Table. This table contains an entry for each exception, which holds the address of the exception handler. Once an exception is encountered, the processor uses the exception number to access the table, to extract the handler address.

Dual stacks

The CR16B features both an Interrupt Stack and a User Stack. The processor uses the Interrupt Stack solely for saving the PC and the PSR during exception processing. This process occurs in hardware, without need for software intervention. The software (user written or through compiler assistance) uses the User Stack for saving the registers and for passing parameters, upon subroutine entry and subroutine calls. This stack is managed by software, with the **PUSH** and **POP** instructions to assist.

This dual stack architecture provides the following benefits:

- The essentials of the processor's state (PC and PSR) are saved correctly on stack, even during nested, non-maskable interrupts; This process does not need to rely on disabling interrupts to allow software to save the PC and PSR on stack
- Separating the Interrupt and User stacks allows for a multi-tasking operating system, which can switch user stacks at task switch, while still maintaining its own PC and PSR on the Interrupt stack
- As the processor saves just the PC and PSR when exception occurs, interrupt latency is kept at a minimum; During exception handling, the software can save only the registers it modifies, thus minimizing interrupt response time, and saving memory.

The exception process

When an exception occurs, the CPU automatically preserves the basic state of the program immediately prior to the occurrence of the exception: A copy of the PC and the PSR is made and pushed onto the Interrupt Stack. Depending on the kind of exception, it restores and/or adjusts the contents of the Program Counter (PC) and the Processor Status Register (PSR). The interrupt exception number is then used to obtain the address of the exception service procedure from the dispatch table, which is then called.

The **RETX** instruction returns control to the interrupted program, and restores the contents of the PSR and the PC registers to their previous status. See the **RETX** instruction on page 5-49.

3.1.2 Interrupt Handling

The CR16B provides three types of interrupts: maskable interrupts, non-maskable interrupt (NMI) and In-System Emulator (ISE).

Maskable interrupts

Maskable interrupts are disabled whenever PSR.E or PSR.I are cleared to 0. PSR.I serves as the global interrupt mask, while PSR.E serves as a local interrupt mask. PSR.E can be easily changed with the **EI** and **DI** instructions (see **EI** instruction on page 5-22 and **DI** instruction on page 5-21). The PSR.E is meant to be used when interrupt-disabling is needed for a short period of time (e.g., when a read-modify-write sequence of instructions, accessing a semaphore, must not be interrupted by another task).

On receipt of a maskable interrupt, the processor determines the exception number (the vector) by performing an interrupt acknowledge bus cycle in which a byte is read from address $00FE00_{16}$. This byte contains a number in the range 16 - 127 (the vector), which is used as an index into the Dispatch Table to find the address of the appropriate interrupt handler. Control is then transferred to that interrupt handler.

Non-maskable interrupt Non-maskable interrupts cannot be disabled; they occur when catastrophic events (such as an imminent power failure) require immediate handling to preserve system integrity. Non-maskable interrupts use vector number 1 in the Dispatch Table. When a non-maskable interrupt is detected, the CR16B performs an interrupt-acknowledge bus cycle to address $00FF00_{16}$, and discards the byte that is read during that bus cycle.

ISE interrupt In-System Emulator (ISE) interrupts cannot be disabled; they temporarily suspend execution when an appropriate signal is activated. ISE interrupts use vector number 15 in the Dispatch Table. When an ISE interrupt is detected, the CR16B performs an interrupt-acknowledge bus cycle to address $00FC00_{16}$, and discards the byte that is read during that bus cycle.

3.1.3 Traps

The CR16B recognizes the following traps:

BPT Trap **Breakpoint Trap**. Used for program debugging. Caused by the **EXCP BPT** instruction.

SVC Trap **Supervisor Call Trap**. Temporarily transfers control to supervisor software, typically to access facilities provided by the operating system. Caused by the **EXCP SVC** instruction.

FLG Trap **Flag Trap**. Indicates various computational exceptional conditions. Caused by the **EXCP FLG** instruction.

DVZ Trap **Division by Zero Trap**. Indicates an integer division by zero. Caused by the **EXCP DVZ** instruction, which can be used by integer division emulation code to indicate this exception.

UND Trap **Undefined Instruction Trap**. Indicates undefined op codes. Caused by an **EXCP UND** instruction, or an attempt to execute any of the following:

- any undefined instruction;
- the **EXCP** instruction when a reserved field is specified in the dispatch table (i.e., reserved trap number).

TRC Trap **Trace Trap.** A TRC trap occurs before an instruction is executed when the PSR.P bit is 1. Used for program debugging and tracing.

DBG Trap **Debug Trap.** A DBG trap occurs as a result of a breakpoint detected by the hardware-breakpoint module, or by an external instruction-execute breakpoint using the tag mechanism through the BRKL line. Used for instruction-execution and data-access breakpoints.

Note: DBG, TRC and BPT traps can also generate an interrupt acknowledge cycle for observability purposes, to alleviate the design of an In-System Emulator. This option can be selected by setting ADBG, ATRC, and ABPT bits respectively in the DCR register. The addresses driven on the bus during these cycles are $00FC02_{16}$, $00FC0C_{16}$ and $00FC0E_{16}$ respectively.

For further information, see Chapter 4, “ADDITIONAL TOPICS”.

3.2 DETAILED EXCEPTION PROCESSING

3.2.1 Instruction Endings

The CR16B checks for exceptions at various points during the execution of instructions. Some exceptions, such as interrupts, are acknowledged between instructions, i.e., before the next instruction is executed. Other exceptions, such as a Breakpoint (BPT) trap, are acknowledged during execution of an instruction. In such a case, the instruction is suspended. See Table 3-1.

If an instruction is suspended, it is not completed, although all other previously issued instructions have been completed. Result operands and flags (except for the PSR.P bit on some traps) are not affected. When an instruction is suspended, the PC and PSR are pushed onto the interrupt stack; The PC saved on the interrupt stack contains the address of the suspended instruction, and the PSR contains the status flags prior to the instruction’s execution.

When an interrupt is detected while a **MULi** instruction is being executed, the **MULi** instruction is suspended. The same is true for the **MULSW**, **MULUW** and **MULSB** instructions. All the other instructions complete execution before an interrupt is serviced.

3.2.2 The Dispatch Table

The CR16B recognizes seven traps, two non-maskable interrupts (NMI and ISE) and up to 112 more maskable interrupts. The dispatch table, pointed to by the concatenated register pair INTBASE == (INTBASEH, INTBASEL), features an entry matching each such exception, containing the exception-handler address. During an exception, the CPU uses the dispatch table to obtain the relevant exception-handler's start address. See Figure 3-1

The CR16B supports either a 16-bit or a 21-bit entry dispatch table, to accommodate exception handlers residing below the 128K boundary, or above it, respectively. In the 16-bit mode, each entry occupies one word of memory, containing bits 1 through 16 of the exception-handler's start address; In the 21-bit mode, each entry occupies two adjacent words in memory, the first holding bits 1 through 16, and the second holding bits 17 through 20 of the exception handler address, right justified. This mode selection is determined by the CFG.ED configuration bit, which also controls the dispatch table's start address:

- (CGF.ED = 0) selects the 16-bit entry dispatch table. Under this selection, INTBASEH is forced to 0, limiting the Dispatch Table's address to the first 64K of memory.
- (CGF.ED = 1) selects the 21-bit entry dispatch table. Under this selection, both INTBASEH and INTBASEL contents are used to calculate INTBASE, thus allowing the Dispatch Table to reside anywhere in memory.
- (CGF.ED = 1), the Dispatch Table's address is less than 2M. Entries in the Dispatch table are then 32 bits wide, but only the lower 21 bits are utilized.

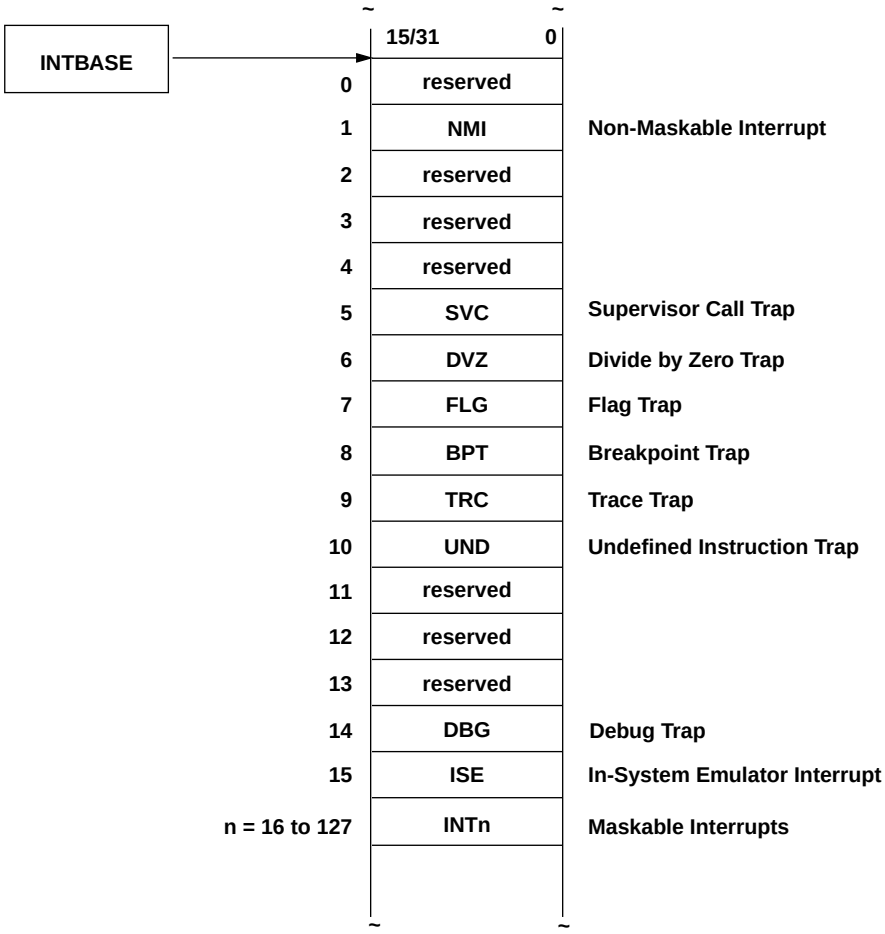


Figure 3-1. Dispatch Table and Jump Table

3.2.3 Acknowledging an Exception

The CR16B performs the following operations in response to interrupt or trap exceptions:

1. Decrements the Interrupt Stack Pointer (ISP) register by 4.

2. Saves the contents the PSR and the PC on the interrupt stack; Since PC is a 21-bit wide register, and the stack is only 16 bits wide (width of memory word in CR16B), the PC and PSR registers are saved on the stack, in two adjacent words, as follows (see Figure 3-2):

- Bits 0 through 11 of the PSR register, and bits 17 through 20 of the PC register, are saved in a single word, at the higher-index address.
- Bits 1 through 16 of the PC register are saved in a single word, at the lower-index address (bit 0 is always 0).

3. Alters PSR by clearing the control bits shown in Table 3-1.

4. For interrupts, displays address and bus-status information during the interrupt acknowledge bus cycle to indicate the type of interrupt encountered as follows:

   If the interrupt is a maskable interrupt, the CPU reads the vector number during this cycle from address $0FE00_{16}$, that is mapped to the Interrupt Control Unit (ICU). This vector is used as the exception number, determining which exception needs to be serviced.

   If the interrupt is a Non Maskable Interrupt (NMI), the CPU performs a read operation from address $0FF00_{16}$ for observability purposes. If the interrupt is an ISE interrupt, the CPU also performs a read operation from address $0FC00_{16}$ for observability purposes.

   In case of non-maskable interrupts or traps, the exception number is set as the vector, to be used while accessing the dispatch table.

5. Reads the double- or single-word entry from the dispatch table at address (INTBASE) + vector $\times$ 4 or (INTBASE) + vector $\times$ 2, according to the value of CFG.ED, 1 or 0 respectively.

   The dispatch table entry is used to call the exception service procedure, and is interpreted as a pointer that is loaded into bits 1 through 16, and, if CFG.ED = 1, also into bits 17 through 20 of the PC.

   Bit 0 of the PC is always cleared. Bits 17 through 20 are cleared only in case of a 16-bit dispatch table (CFG.ED = 0). See Figure 3-3.

   The pointer is stored in the dispatch table in little-endian format. For a 32-bit entry, the lower address word contains address bits 1 through 16, and the upper address word contains address bits 17 through 20.

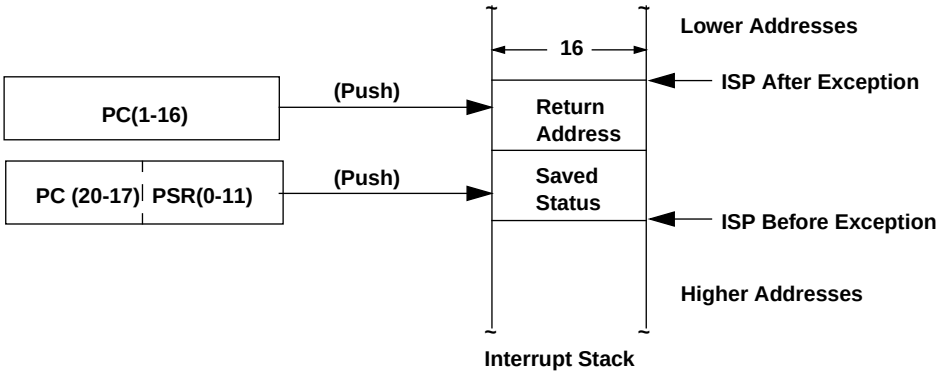
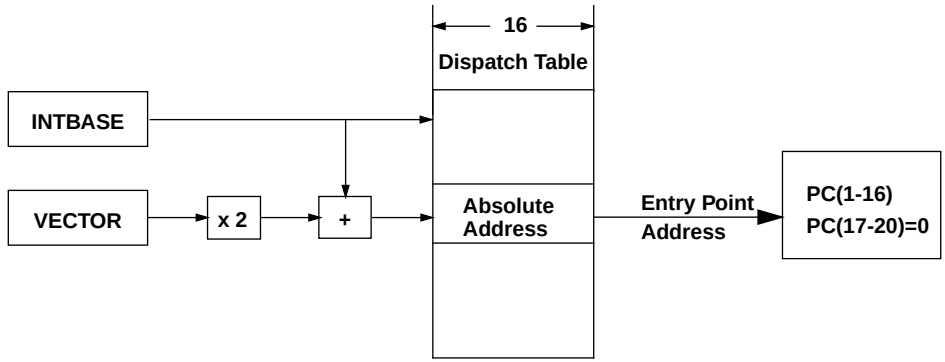


Figure 3-2. Saving PC and PSR Contents During an Exception Processing Sequence

16-bit Dispatch Table (Small Memory Model): CFG.ED = 0



32-bit Dispatch Table (Large Memory Model): CFG.ED = 1

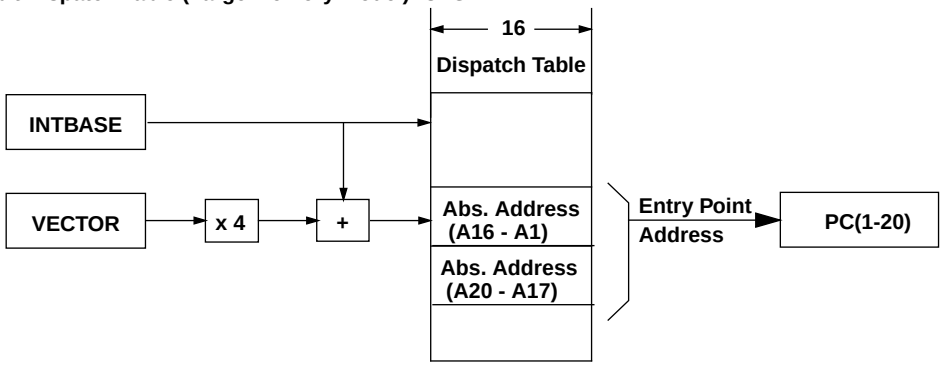


Figure 3-3. Transfer of Control During an Exception Acknowledge Sequence

Exception processing summary

Table 3-1 summarizes how each type of exception is handling.

Table 3-1. Summary of Exception Processing

Exception	Instruction Completion Status	PC Saved	Cleared PSR Bits	
			Before Saving PSR	After Saving PSR
Interrupt	Before start of instruction	Next	None	I P T
Interrupt during execution of **MULi**, **MULSB**, **MULSW**, **MULUW**	Suspended	Current	P[a]	I P T
BPT, DVZ, FLG, SVC	Suspended	Current	None[b]	P T
UND	Suspended	Current	None[b]	P T
TRC	Before start of instruction	Next	P	P T
DBG	Before Instruction	Next	None	I P T

a. The PSR.P bit is cleared if an interrupt is acknowledged before a **MULi**, **MULSB**, **MULSW** or **MULUW** instruction is completed, to prevent a mid-instruction trace trap upon return from the exception service procedure.

b. The trap handler may need to clear the P bit of the PSR, which is saved on stack, to prevent a redundant trace exception, in case Trace trap is used in conjunction with these exceptions. For more information, refer to “Retry execution of a suspended instruction” on page 3-10 and “Clearing PSR.P bit on Interrupt Stack” on page 4-2.

3.2.4 Exception Service Procedures

After the CR16B acknowledges an exception, control is transferred to the appropriate exception service procedure. The TRC trap is disabled (the PSR.P and PSR.T bits are cleared). Maskable interrupts are also disabled (the PSR.I bit is cleared) for a service procedure called in response to an interrupt or a DBG trap.

At the beginning of each instruction, the PSR.T bit is copied into the PSR.P. If PSR.P is still set at the end of the instruction, a TRC trap is executed before the next instruction.

To complete a suspended instruction, program the exception service procedure either to simulate the suspended instruction, or to retry execution of the suspended instruction.

Simulate a suspended instruction

The exception service procedure can use software to simulate execution of the suspended instruction. After it calculates and writes the results of the suspended instruction, it should modify the flags in the copy of the PSR which were saved on the interrupt stack, and update the PC saved on the interrupt stack to point to the next instruction to be executed.

The exception service procedure can then execute the **RETX** instruction, and the CR16B begins executing the instruction following the suspended instruction. For example, when an Undefined Instruction Trap (UND) occurs, software can be used to perform the appropriate corrective actions.

Retry execution of a suspended instruction

The suspended instruction can be retried after the exception service procedure has corrected the trap condition that caused the suspension.

In this case, the exception service procedure should execute the **RETX** instruction at its conclusion; then the CR16B retries the suspended instruction. A debugger takes this action when it encounters an **EXCP BPT** instruction that was temporarily placed in another instruction's location in order to set a breakpoint. In this case, exception service procedures should clear the PSR.P bit to prevent a TRC trap from occurring again.

3.2.5 Returning From Exception Service Procedures

Exception service procedures perform actions appropriate for the type of exception detected. At their conclusion, service procedures execute the **RETX** instruction to resume executing instructions at the point where the exception was detected. For more information about the **RETX** instruction, see "RETX Return from Exception" on page 5-48.

3.2.6 Priority Among Exceptions

The CR16B checks for specific exceptions at various points while executing an instruction (see Figure 3-4).

If several exceptions occur simultaneously, the CR16B responds to the exception with the highest priority, accordingly.

If several maskable interrupts occur simultaneously, the Interrupt Control Unit (ICU) determines the highest priority interrupt, and requests the CR16B to service this interrupt.

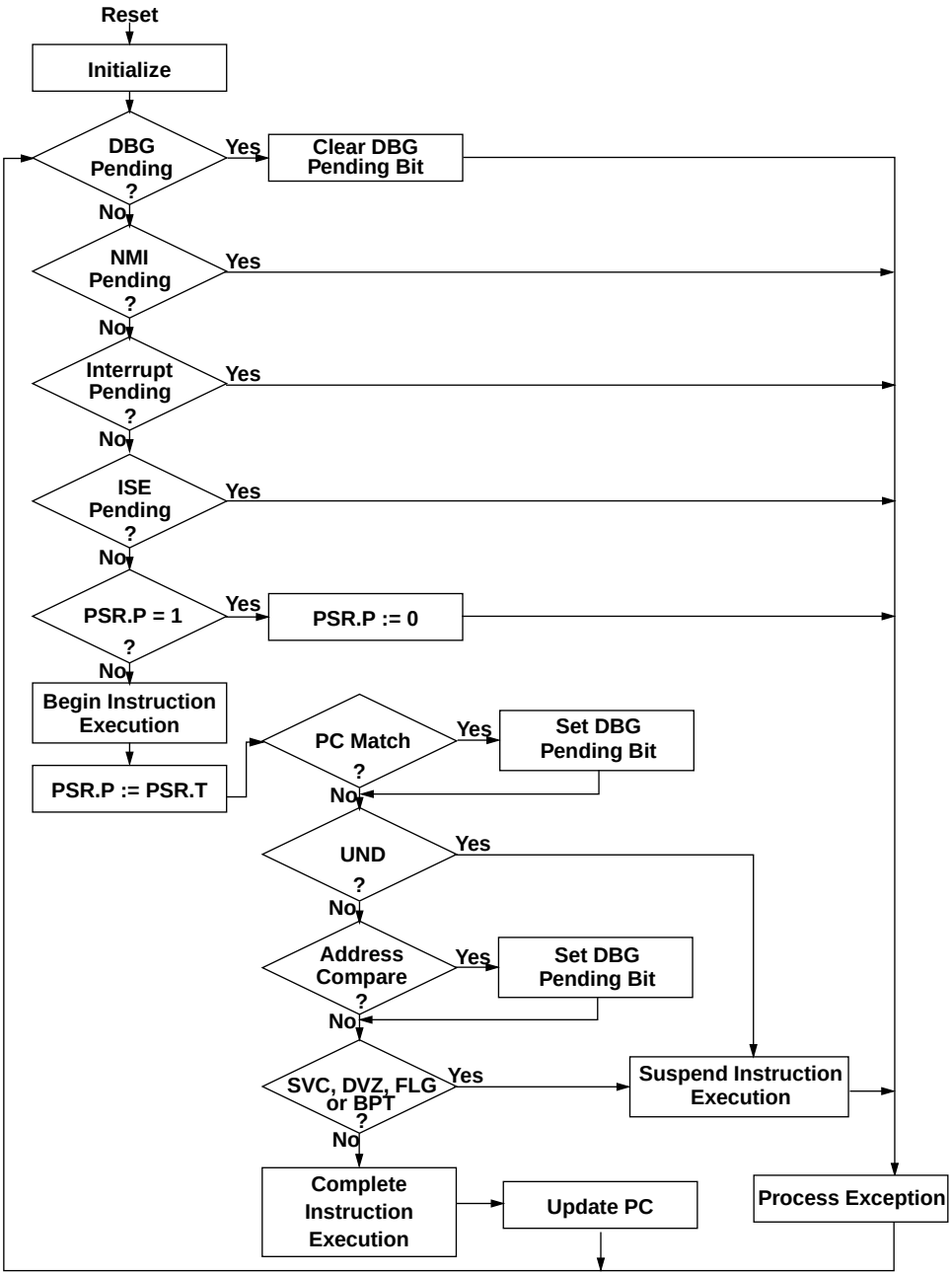


Figure 3-4. Exception Processing Flowchart

Before executing an instruction

Before executing an instruction, the CR16B checks for pending DBG traps, interrupts and trace traps, in that order. It responds to the interrupts in order of descending priority (i.e., first non-maskable interrupts, then maskable interrupts and lastly, ISE interrupts.

If no interrupt is pending, and PSR.P is 1 (i.e., a trace trap is pending), then the CR16B clears PSR.P and processes the trace trap.

If no trace trap or interrupt is pending, the CR16B begins executing the instruction by copying PSR.T to PSR.P. While executing an instruction, the CR16B may detect a trap.

During execution of an instruction

First, the CR16B checks for an undefined instruction (UND) trap; then it looks for any of the following mutually exclusive traps: SVC, DVZ, FLG or BPT. The responds to the first trap it detects by suspending the current instruction and executing the trap.

If an undefined instruction is detected, then no data references are performed for the instruction.

If an interrupt becomes pending during execution of the **MULi**, **MULSW**, **MULSB** and **MULUW** instructions the CR16B clears PSR.P and the DBG pending bit (if any) to 0 and responds to the requested interrupt. The different **MUL** instructions, when interrupted, do not modify any external register or variable, until completion.

If no exception is detected while the instruction is executing, the instruction is completed (i.e., values are changed in registers and memory, except for PSR.P, which was changed earlier) and the PC is updated to point to the next instruction.

While the CPU executes an instruction, it checks for enabled debug conditions. If the CPU detects an enabled address-compare, or a PC-match debug condition, a Debug (DBG) trap remains pending until after the instruction is completed.

3.2.7 Nested Interrupts

A nested interrupt is an interrupt that occurs while another interrupt is being serviced. Since the PSR.I bit is automatically cleared before any interrupt is serviced (see Table 3-1), nested maskable interrupts are not serviced by default. However, the Exception Service Procedure can explicitly allow nested maskable interrupts at any point, by setting the PSR.I bit using a **LPR** instruction. However, the handler must verify that the PSR.E bit is set as well, for interrupts to occur. In this case, pending maskable interrupts are serviced normally even in the middle of the currently executing Exception Service Procedure.

It is possible to enable nesting of specific maskable interrupts inside a certain Exception Service Procedure. This is done by programming the Interrupt Control Unit (ICU) to mask the undesired interrupt sources, during the execution of the Exception Service Procedure. This should be done before the PSR.I bit is set.

Nested Non Maskable Interrupt (NMI) and nested ISE interrupt are always serviced.

The interrupt nesting level is limited only by the amount of memory that is available for the interrupt stack.

3.3 RESET

A reset occurs when the appropriate signal is activated. Reset must be used at power-up to initialize the CR16B.

As a result of a reset operation:

- All instructions currently being executed are terminated.
- Results and flags normally affected by the terminated instruction are unpredictable.
- The results of instructions, whose execution started but did not yet end, may not be written to their destinations.
- Any pending interrupts and traps are eliminated.

Upon reset, the following operations are executed:

- The current values of bits 1 through 16 of the PC are stored in R0.
- The current value of the PSR, bits 0 through 11, is concatenated with the current values of bits 17 through 20 of the PC, and the result is stored in R1. (See Figure 3-2.)
- The following internal registers are cleared to 0: PC, CFG, DCR and PSR, except for PSR.E, which is set to 1.
- DCR.DBGL is cleared to 0 .

After reset, the processor begins normal execution at memory location 0, and the reserved bits in these registers, and the contents of all other registers, are unpredictable.

Chapter 4
ADDITIONAL TOPICS

4.1 DEBUGGING SUPPORT

The following CR16B features make program debugging easier.

- Instruction Tracing
- Soft Break Generation by Breakpoint Instruction (**EXCP BPT**)
- Breakpoint Triggered by a PC Address Match
- Breakpoint Triggered by a Compare-Address Match (Data transfer)
- Instruction-execute breakpoint triggered by an external source during the Fetch.
- ISE Support

The Processor Status Register (PSR), the Debug Control Register (DCR), the Debug Status Register (DSR), and the Compare-Address Register (CAR) control and monitor the debug and trace features. Loading and storing these registers is done through the **LPR** and **SPR** instructions. Accesses to the 21-bit CAR register are done by reading and writing its low and high 16-bit portions (CARL and CARH).

4.1.1 Instruction Tracing

Instruction tracing can be used during debugging to single-step through selected portions of a program. The CR16B uses two bits in the PSR to enable and generate trace traps. Tracing is enabled by setting the T bit in the PSR register.

During the execution of each instruction, the CR16B copies the PSR.T bit into the PSR.P (trace pending) bit. Before beginning the next instruction, the CR16B checks the PSR.P bit to determine whether a Trace (TRC) trap is pending. If PSR.P is 1, i.e., a trace trap is pending, the CR16B generates a trace trap before executing the instruction.

For more information on the different exception priorities, see Figure 3-4, “Exception Processing Flowchart” on page 3-11.

For example, if an Undefined Instruction (UND) trap is detected while tracing is enabled, the TRC trap occurs after execution of the **RETX** instruction that marks the end of the UND service procedure. The UND service procedure can use the PC value, saved on top of the interrupt stack, to determine the location of the instruction. The UND service procedure is not affected, whether instruction tracing was enabled or not.

Clearing PSR.P bit on Interrupt Stack

Trap handlers for exceptions which cause instruction suspension (UND, BPT, DVZ, FLG and SVC), may need to clear the copy of the PSR.P bit, saved on the interrupt stack, before resuming execution. This must be done if the exception service replaces the exception invocation instruction with code for execution, and attempts to re-execute that location, according to the saved PC on stack. Otherwise, when attempting to re-execute that location, the processor will perform a redundant trace exception before executing the said instruction, since the PSR.P bit is set in the restored PSR.

Note the following:

- **LPR** (on PSR) and **RETX** instructions cannot be reliably traced because they may alter the PSR.P bit during their execution.
- If instruction tracing is enabled while the **WAIT** or **EIWAIT** instruction is executed, a trace trap occurs after the next interrupt, when the interrupt service procedure returns.

The breakpoint instruction

Debuggers can use the breakpoint instruction (**EXCP BPT**) to stop the execution of a program at specified instructions, to examine the status of the program. The debugger replaces these instructions with the breakpoint instruction. It then starts the program execution. When such an instruction is reached, the breakpoint instruction causes a trap, which enables the debugger to examine the status of the program at that point.

PC match

The CR16B provides a hardware breakpoint register to allow setting of breakpoints in ROM, as well as in RAM.

A PC match is detected when both DCR.DEN and DCR.PC are set to 1, and the address of the instruction equals the value specified in the 21-bit wide CAR register. Accesses to the CAR register are done by reading and writing its low and high 16-bit portions (CARL and CARH). Bits 21 through 31 are reserved, and must be set to 0.

When a PC match is detected, a debug trap (DBG) is held pending until after the instruction is completed. The cause of the DBG trap is indicated by setting DSR.BPC to 1, and clearing DCR.DEN to 0.

The PC match and the Compare-Address match are mutually exclusive. The PC match takes precedence (i.e., DCR.PC set to 1 disables the Compare-Address match).

External tag on fetch

The CR16B provides an input (BRKL) which allows an external device to detect a breakpoint condition, and tag an incoming instruction during the fetch stage.

This line is sampled during the first word fetch of each instruction, during the last cycle of the data transfer, in parallel to data_rd sampling by the core.

The tag is transferred into the decode unit, and just before the instruction is due for execution, the external break conditions are evaluated. If DCR.EFB is set, and the tag bit for the instruction is set, a Debug (DBG) trap is inserted, before the core starts the original instruction. The cause of the DBG trap is indicated by clearing the DCR.EFB bit, and setting the DSR.EXF bit.

The exception saves a PC which matches that of the tagged instruction, thus ensuring correct identification of the breakpoint by the exception handler.

If the pin count on a part is limited, it is recommended that its line be multiplexed with the PLI line in the system interface.

4.1.2 Compare-Address Match

A breakpoint may also result from a data access. A compare-address match is detected when a memory location is either read or written. The word address used for the comparison is specified in bits 1 through 20 of the Compare-Address Register (CAR). The bytes within the word to be compared are specified by DCR.CBE0,1. All the bytes accessed in the load or store operation are compared.

Both DCR.CRD and DCR.CWR can enable a compare-address match for read and write references. The CR16B examines the compare-address condition for all data reads and writes, interrupt-acknowledge bus cycles, and memory references for exception processing. A compare-address match is enabled for read accesses whenever DCR.DEN and DCR.CRD are set to 1, and DCR.PC is cleared to 0. A compare-address match is enabled for write accesses whenever DCR.DEN and DCR.CWR are set to 1, and DCR.PC is cleared to 0.

When the CR16B detects a compare-address match, a Debug (DBG) trap is held pending until after the instruction is completed. The cause of the DBG trap is indicated by setting either DSR.BWR or DSR.BRD to 1, and DCR.DEN is cleared to 0.

4.1.3 Checking for Debug and Breakpoint Conditions

The Debug Status Register (DSR) indicates debug and breakpoint conditions that have been detected. When the CPU detects an enabled debug condition, it sets the appropriate bits in the DSR to 1. Bits 0 through 3 of the DSR are cleared to 0 at reset. In addition, software must clear all the bits in the DSR when appropriate.

The format of the DSR is shown below.

15 4	3	2	1	0
reserved	EXF	BRD	BWR	BPC

BPC **Program Counter Bit**. Set when a PC match is detected.

BWR **Write Bit**. Set when a compare-address match is detected for a data write.

BRD **Read Bit**. Set when a compare-address match is detected for a data read.

EXF **Fetch-tagged Execution Breakpoint Bit**. Set when an instruction which was tagged during the fetch stage reaches the execution phase.

4.1.4 Controlling the Debug and In-System-Emulator Options

Breakpoint Control

The Debug Control Register (DCR) controls the compare-address match, external tag on fetch, and PC match debug, options. These options are enabled and controlled with the appropriate bits in the DCR. When a bit is set to 1, the condition it controls is enabled; otherwise, it is disabled. A DBG trap may be triggered by a PC match, by an external tag on fetch reaching execution, or by a compare-address match. The cause of a DBG trap is indicated in the DSR register. The DSR register is cleared to 0 at reset.

Exception Addressing for In System Emulators

The CR16B includes a new option which makes it easier for an external In System Emulator to detect and identify relevant exceptions.

On ISE, DBG, TRC and BPT exceptions, if the appropriate bit in DCR is set (AISE, ADBG, ATRC, ABPT respectively), the core assumes an INTBASE of 0, and assumes a 32-bit entry dispatch table for these exceptions, regardless of the value of CFG.ED:

Entry Address = 0 + VECTOR# x 4

Additionally, if the appropriate control bit is set, an interrupt acknowledge cycle precedes the dispatch table access for DBG, TRC and BPT, for observability purposes.

Debug Control Register (DCR)

The format of the DCR is shown below.

15	14	13	12	11	10	9	8	7	6	5	4	3 2	1 0
DBGL	res	AISE	ADBG	ATRC	ABPT	res	EFB	DEN	CRD	CWR	PC	res	CBE

CBE **Compare Byte Enable.** These bits (0 through 1) specify which corresponding bytes of an aligned word may trigger a breakpoint during address or PC comparison.

PC **PC Match** or **Compare-Address Match**. Enables generation of a DBG trap on a PC match, when it is set to 1. When it is cleared to 0, it enables generation of a DBG trap on a compare-address match.

CWR **Compare-Address on Write**. When set to 1, enables address comparison for write operations.

CRD **Compare-Address on Read**. When set to 1, enables address comparison for read operations.

DEN **Debug Condition Enable**. Enables either the compare-address match or PC match condition, depending on the value of the PC bit.
DEN is cleared to 0 when PC match is achieved, and DSR.BPC is set to 1, or when address-match is achieved, and DSR.BRD or DSR.BWR is set to 1.

EFB **Enable Tag-at-Fetch Breakpoint**. Enable breaking on an instruction tagged during the fetch phase by an external device.
EFB is cleared to 0 when the tagged instruction reaches execution, and DSR.EXF is set to 1.

ABPT **Alternate behavior on BPT**. Perform an Interrupt-Acknowledge cycle for BPT and calculate dispatch table entry based on INTBASE = 0, and 32-bit entries.

ATRC **Alternate behavior on TRC**. Perform an Interrupt-Acknowledge cycle for TRC and calculate dispatch table entry based on INTBASE = 0, and 32-bit entries.

ADBG **Alternate behavior on DBG**. Perform an Interrupt-Acknowledge cycle for DBG and calculate dispatch table entry based on INTBASE = 0, and 32-bit entries.

AISE **Alternate behavior on ISE**. Calculate dispatch table for ISE based on INTBASE = 0, and 32-bit entries. ISE always performs an Interrupt-Acknowledge cycle.

DBGL **Debug-In-System-Emulation Support Lock**: A set-once bit (cleared only by reset), locking the ABPT, ATRC, ADBG, AISE bits in DCR, and the ON bit in DBGCFG.

Note: DBGCFG is a chip-level debug-configuration register, used to control the behavior of the whole part, for debugging purposes. This register resides outside the core, and is described in each part-specific specification, or user manual.

4.1.5 In-System Emulator (ISE)

The CR16B core provides the following to support the development of real-time In-System Emulator (ISE) equipment and Application Development Boards (ADBs).

- Status signals that indicate when an instruction in the execution pipeline is completed.
- Status signals that indicate the type of each bus cycle, e.g., fetch.
- Status signals that indicate when there is a non-sequential fetch.
- An ISE interrupt signal.
- An interrupt acknowledge cycle for ISE interrupt.
- An interrupt acknowledge cycle for the DBG, TRC and BPT exceptions, programmable by the DCR register.
- A BRKL line for accurate external breakpoints on instruction execution, by tagging instructions during the fetch cycle.
- Forcing of INTBASE to 0 for the ISE, DBG, TRC and BPT exceptions, programmable by the DCR register.

- A set-only lock bit in DCR, protects the ABPT, ADBG, ATRC, AISE bits in DCR and DBGCFG.ON from erroneous clearing by user software.
- A special bus status signal during exception handling, that indicates that the dispatch table is being read.
- A special early bus status signal (available only on some system interfaces) indicating that either core code fetch, or core dispatch-table read is being executed.
- Upon reset, the CR16B stores the contents of the PSR and bits 17 through 20 of the PC, in R1, and the contents of bits 1 through 16 of the PC register in R0.

4.2 INSTRUCTION EXECUTION ORDER

The CR16B has four operating states in which instructions may be executed and exceptions may be processed. They are:

- Reset
- Executing Instructions
- Processing Exception
- Waiting for Interrupt

Figure 4-1 shows these states, and the transitions between them.

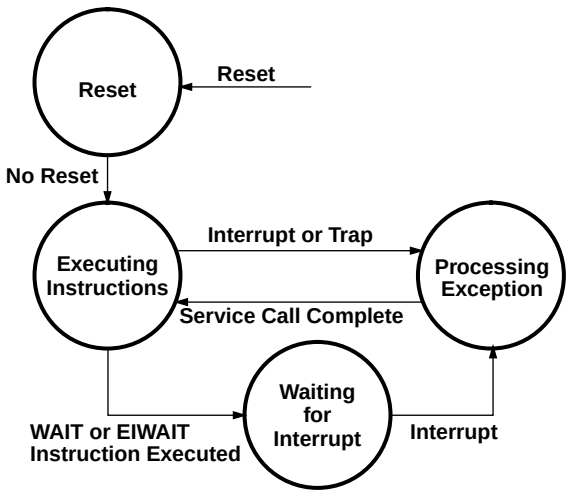


Figure 4-1. CR16B Operating States

Reset

When the reset input signal is activated, the CR16B enters the reset state. In this state, the contents of certain dedicated registers are initialized, as detailed in “Reset” on page 3-13.

Executing instructions

When the reset signal is deactivated, the CR16B enters the executing-instructions state. In this state, the CR16B executes instructions repeatedly until an exception is recognized, or a **WAIT**, or an **EIWAIT**, instruction is executed.

Processing exception

When an exception is recognized, the CR16B enters the processing exception state in which it saves the PC and the PSR contents. The processor then reads an absolute address from the Interrupt Dispatch Table and branches to the appropriate exception service procedure. See “The exception process” on page 3-2 for more information.

To process maskable interrupts, the CR16B also reads a vector value from an Interrupt Control Unit (ICU).

After successfully completing all data references required to process an exception, the CR16B reverts to the executing instructions state.

Waiting for interrupt

When a **WAIT** or an **EIWAIT** instruction is executed, the CR16B enters the wait for interrupt state in which it is idle. When an interrupt is detected the processor enters the processing exception state.

4.2.1 The Instruction Pipeline

The operations for each instruction are not necessarily completed before the operations of the next instruction begin. The CR16B can overlap operations for several instructions, using a pipelined technique to enhance its performance. While the CR16B is fetching one instruction, it can simultaneously decode a second instruction and calculate results for a third instruction (see Figure 4-2).

In most cases, pipelined instruction execution improves performance while producing the same results as strict sequential instruction execution. Under certain circumstances, however, the effects of this performance enhancement are visible to system software and hardware as differences in the order of memory references performed by the CR16B. See the explanation below.

Instruction fetches

The CR16B fetches an instruction only after all previous instructions have been completely fetched. It may, however, begin fetching the instruction before all of the source operands have been read, and before the results have been written for previous instructions.

Operands and memory references

The source operands for an instruction are read only after all data reads and data writes, in previous instructions, have been completed. Figure 4-2 shows this process, and the order of precedence of memory reference for two consecutive instructions. The arrows indicate the order of precedence between operations in an instruction, and between instructions.

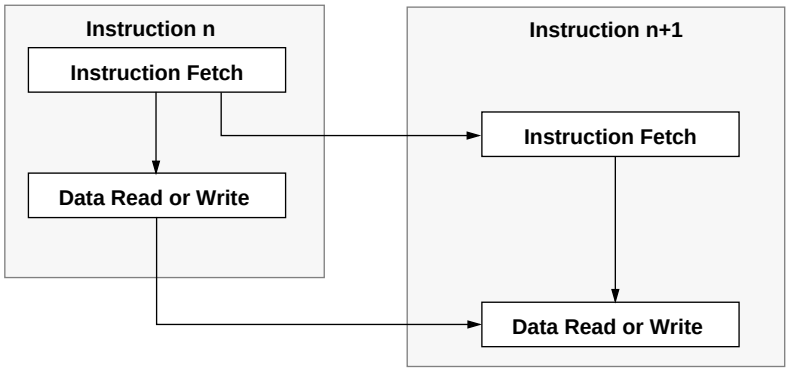


Figure 4-2. Memory References for Consecutive Instructions

Overlapping operations

As a consequence of overlapping the operations for several instructions, the CR16B may fetch an instruction, but not execute it (e.g., if the previous instruction causes a trap). The CR16B reads source operands, and writes destination operands for executed instructions only.

Dependencies

The CR16B does not check for dependencies between the fetching of the next instruction and the writing of the results of the previous instructions. Therefore, special care is required when executing self-modifying code.

4.2.2 Serializing Operations

The CR16B serializes instruction execution after processing an exception. This means that it finishes writing all the results of the preceding instructions to a destination, before it fetches the first instruction of the interrupt service procedure. This fetch is non-sequential.

The CR16B also serializes instruction execution after executing the following instructions: **LPR**, **SPR**, **RETX**, and **EXCP**.

Chapter 5
INSTRUCTION SET

This chapter describes each of the CR16B instructions, in detail.

5.1 INSTRUCTION DEFINITIONS

The name of each operand appears in bold italics, and indicates its use. In addition, the valid addressing modes, access class and length are specified for each operand. The addressing mode may be: *reg* (register), *regp* (register-pair), *procreg* (processor register), *imm* (immediate), *abs* (absolute) *rel* (relative) or *far* (far relative). The access class may be *read*, *write*, *rmw* (read-modify-write), *addr* (address) or *disp* (displacement). The access class is followed by a data length attribute specifier. See Figure 5-1.

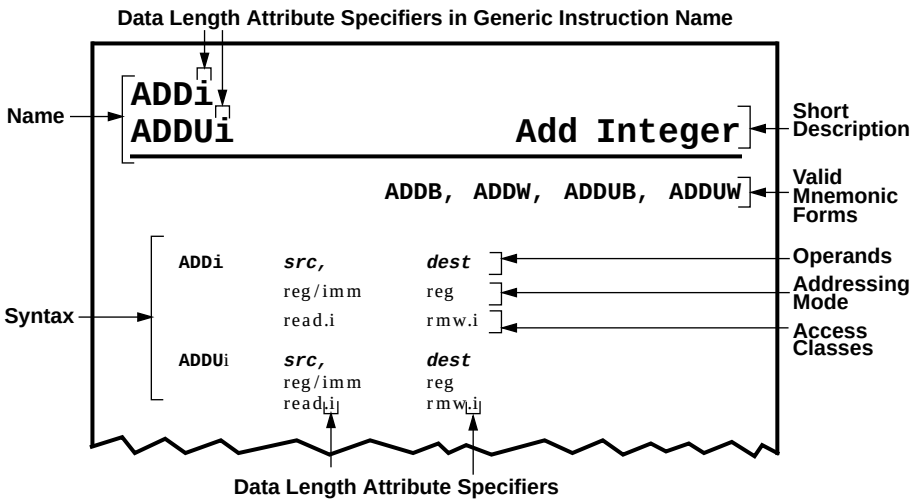


Figure 5-1. Instruction Header Format

The data length attribute specifier specifies how the operands are interpreted, and represents a character that is incorporated into the name of the actual instruction. The **i** specifier stands for a **B** (byte) or **W** (word) in the actual instruction name. In the access class, the **L** (long) specifier stands for the long, 21-bit address/displacement, needed for the CR16B. In the **BR/BRCond** instructions, the mnemonics for the CR16B 21bit displacement instructions has been unified with that of the 16A 17-bit displacement instructions. Machine-code encoding is according to the memory model selected for the compiler/assembler (Large memory model is 21-bit displacement encoded; Small memory model is 17-bit displacement encoded).

Each instruction definition is followed by a detailed example of one or more typical forms of the instruction. In each example, all the operands of the instruction are identified, both those explicitly stated in assembly language and those that are implicitly affected by the instruction.

For each example, the values of operands before and after execution of the instruction are shown. Often the value of an operand is not changed by the instruction. When the value of an operand changes, its field is highlighted, i.e., its box is grey. See Figure 5-2.

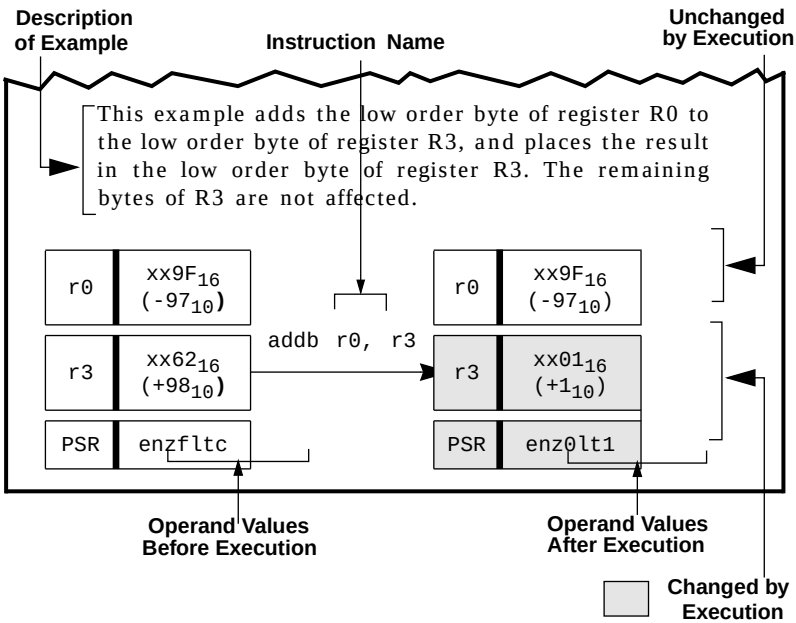


Figure 5-2. Instruction Example Format

The least significant digit of the least significant byte is the right-most digit. Values are expressed in terms of a radix in a subscript to the value.

An **x** represents a binary digit or a hexadecimal digit (4 bits) that is either ignored or unchanged.

5.2 DETAILED INSTRUCTION LIST

The following pages describe in detail the instruction set.

ADDi
ADDUi

Add Integer

ADDB, ADDW, ADDUB, ADDUW

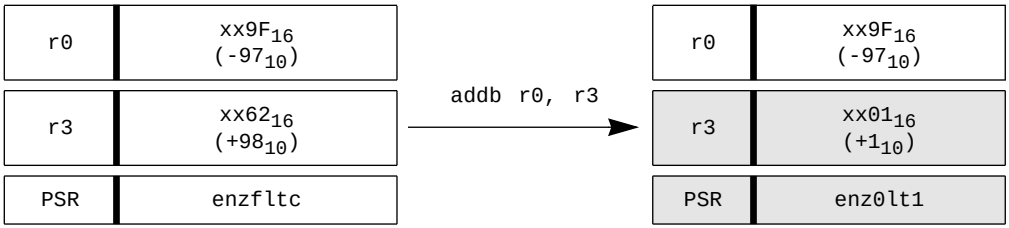
ADDi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i
ADDUi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **ADDi** and **ADDUi** instructions add the ***src*** and ***dest*** operands, and place the result in the ***dest*** operand.

Flags: During execution of an **ADDi** instruction, PSR.C is set to 1 on a carry from addition, and cleared to 0 if there is no carry. PSR.F is set to 1 on an overflow from addition, and cleared to 0 if there is no overflow. PSR flags are not affected by the **ADDUi** instruction.

Traps: None

Example: Adds the low-order byte of register R0 to the low-order byte of register R3, and places the result in the low-order byte of register R3. The remaining bytes of R3 are not affected.



ADDCi Add Integer with Carry

ADDCB, ADDCW

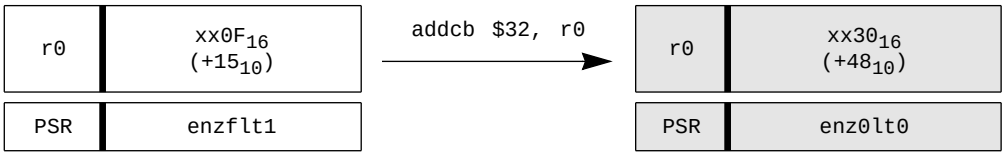
ADDCi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **ADDCi** instructions add the ***src*** operand ***dest*** operand and the PSR.C flag, and place the sum in the ***dest*** operand.

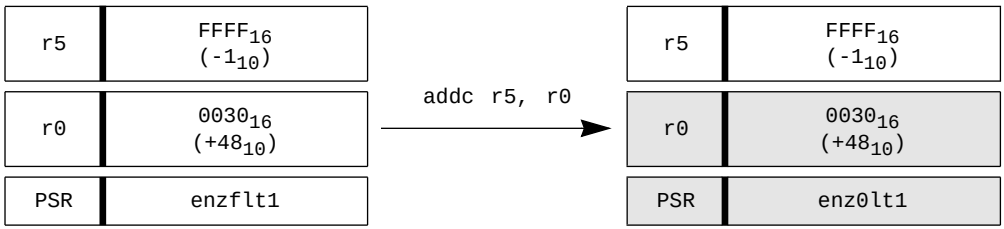
Flags: PSR.C is set to 1 if a carry occurs, and cleared to 0 if there is no carry. PSR.F is set to 1 if an overflow occurs, and cleared to 0 if there is no overflow.

Traps: None

Examples: 1. Adds 32, the low-order byte of register R0, and the PSR.C flag contents, and places the result in the low-order byte of register R0. The remaining bytes of register R0 are unaffected.



2. Adds the contents of registers R5 and R0, and the contents of the PSR.C flag, and places the result in register R0.



ANDi — Bitwise Logical AND

ANDB, ANDW

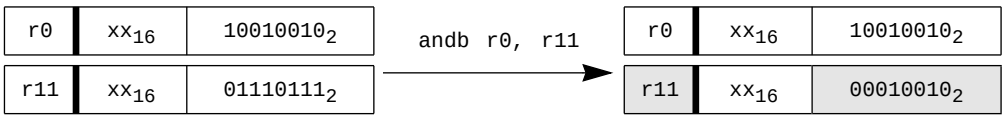
ANDi	*src*,	*dest*
	reg/imm	reg
	read.i	rmw.i

The **ANDi** instructions perform a bitwise logical AND operation on the ***src*** and dest operands, and place the result in the ***dest*** operand.

Flags: None

Traps: None

Example: ANDs the low-order bytes of registers R0 and R11 and places the result in the low-order byte of register R11. The remaining byte of register R11 is unaffected.



ASHUB, ASHUW

ASHUi	***count***,	***dest***
	reg/imm	reg
	read.B	rmw.i

The **ASHUi** instructions perform an arithmetic shift on the ***dest*** operand as specified by the ***count*** operand. Both operands are interpreted as signed integers.

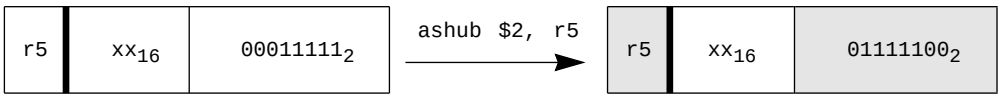
The sign of ***count*** determines the direction of the shift. A positive ***count*** specifies a shift to the left; a negative ***count*** specifies a shift to the right. The absolute value of the ***count*** specifies the number of bit positions to shift the ***dest*** operand. The ***count*** operand value must be in the range –7 to +7 if **ASHUB** is used; and in the range −15 to +15 if **ASHUW** is used. Otherwise, the result is unpredictable.

If the shift is to the left, high-order bits (including the sign bit) shifted out of ***dest*** are lost, and low-order bits emptied by the shift are filled with zeros. If the shift is to the right, low-order bits shifted out of ***dest*** are lost, and high-order bits emptied by the shift are filled from the original sign bit of ***dest***.

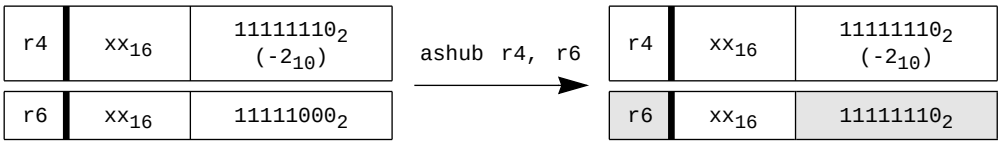
Flags: None

Traps: None

Examples: 1. Shifts the low-order byte of register R5 two bit positions to the left. The remaining byte of register R5 is unaffected.



2. Reads a byte from register R4. Based on this value, it shifts the low-order byte of register R6 accordingly. The remaining byte of register R6 is unaffected.



BEQ, BNE, BCS, BCC, BHI, BLS, BGT, BLE, BFS, BFC, BLO, BHS, BLT, BGE

Bcond ***dest***
imm
disp

If the condition specified by **cond** is true, the **Bcond** instruction causes a branch in program execution. Displacement interpretation varies depending on compilation in small or large memory model, and the size of the displacement. Program execution continues at the location specified by ***dest***, sign extended to 21 bits, plus the current contents of the Program Counter. If the condition is false, execution continues with the next sequential instruction. Table "BR/BRcond Target Addressing Methodology" on page 5-9 summarizes the different addressing calculations. See "BR/Bcond" on page 5-65.

cond is a two-character condition code that describes the state of a flag, or flags, in the PSR register. If the flag(s) are set as required by the specified **cond**, the condition is true; otherwise, the condition is false. The following table describes the possible **cond** codes and the related PSR flag settings:

cond Code	Condition	True State
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0

cond Code	Condition	True State
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal	Z or N flag is 1

The following table describes the displacement extension and address wrap-around calculation, with relation to the displacement size and memory-model. This methodology keeps backwards compatibility with most CR16A code in the small memory model, while allowing both usage of the short displacement and coverage of the whole 2 Mbytes with just 21 bits displacement in the large memory model.

Table 5-1. BR/BRcond Target Addressing Methodology

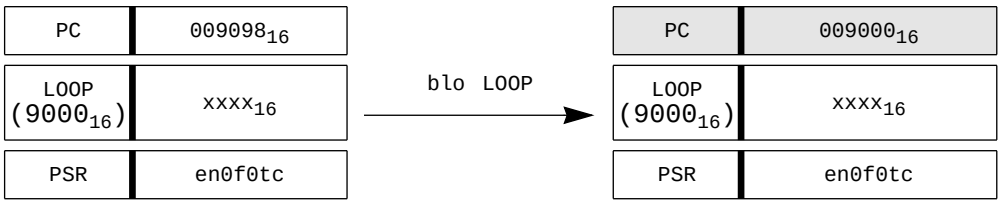
Displacement Needed (signed)	Memory Model	Allowed Address Range	Address Calculation
9 bits	Small/Large	0 - 2M	PC <- Truncate_21[a](PC + sext21[b](disp))
17 bits	Small	0 - 128K	PC <- Truncate_17[c](PC + sext21[b](disp))
21 bits	Large	0 - 2M	PC <- Truncate_21[a](PC + disp)

a. Truncate_21 means the whole calculation is performed at 21 bits, creating a 2 Mbyte wrap around. Address bit 0 is cleared to 0.
b. sext21: sign extend the displacement to 21 bits.
c. truncate_17: Truncate the address calculation result at 17 bits - 128K byte wrap around. Address bits 17 through 20 and bit 0 are cleared to 0.

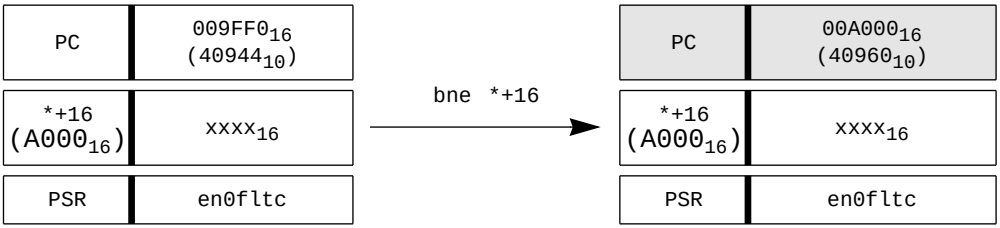
Flags: None

Traps: None

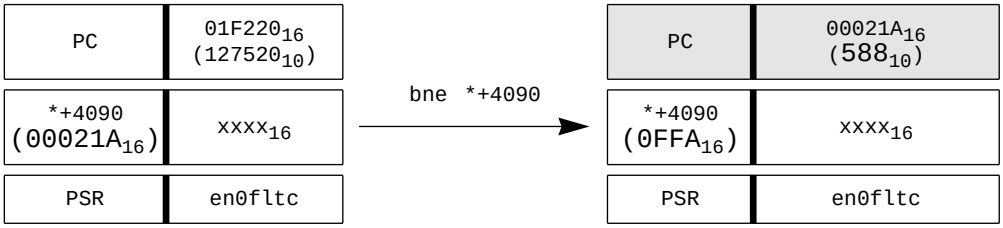
Examples: 1. Passes execution control to the instruction labeled LOOP by adding $1FFF68_{16}$ to the PC register, with a wrap-around at 2M (21-bit truncation at 200000_{16}), if the PSR.Z and PSR.L flags are 0.



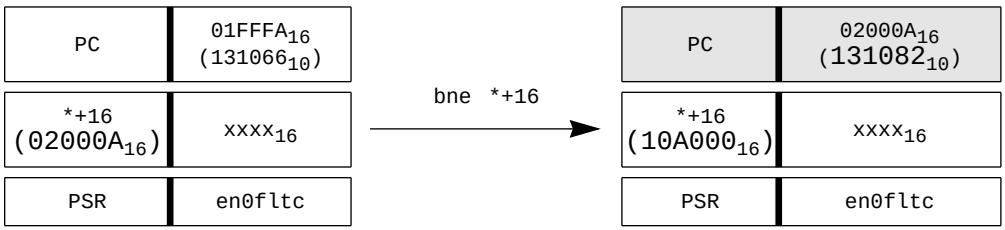
2. Passes execution control to a non-sequential instruction if the PSR.Z flag is 0. The instruction passes execution control by adding 16 to the PC register.



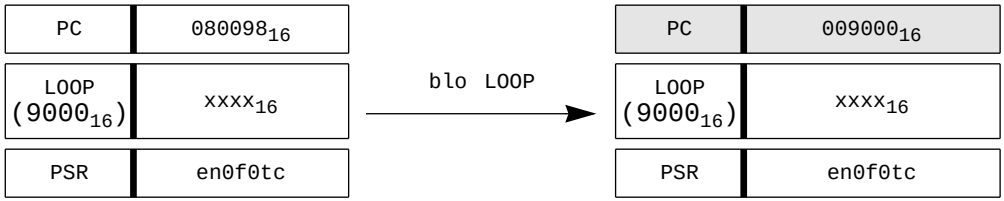
3. Passes execution control to a non-sequential instruction if the PSR.Z flag is 0. The instruction passes execution control by adding 4090 to the PC register, wrapping at the 128K boundary (small memory model).



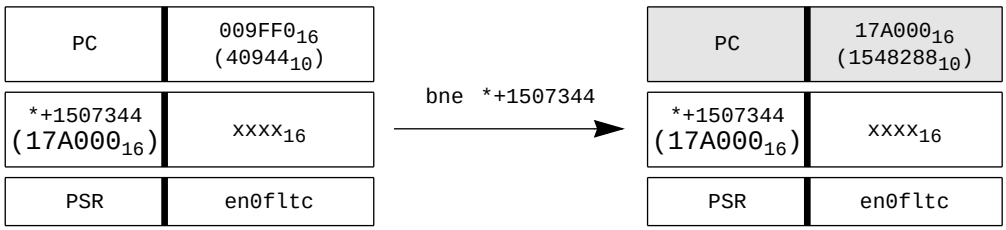
4. Passes execution control to a non-sequential instruction if the PSR.Z flag is 0. The instruction passes execution control by adding 16 to the PC register, crossing the 128K boundary.



5. Passes execution control to the instruction labeled **LOOP** by adding $188F68_{16}$ to the PC register (wrap at 200000_{16}), if the PSR.Z and PSR.L flags are 0. (large memory model)



6. Passes execution control to a non-sequential instruction if the PSR.Z flag is 0. The instruction passes execution control by adding 1507344 to the PC register. (large memory model)



BAL

BAL	***link***,	***dest***
	reg/regp	imm
	write.W/L	disp

The address (bits 16 - 1 for small memory model, bits 20 - 1 for large memory model) of the next sequential instruction is first stored in the register for small memory model, or in the register pair for large memory model, specified as the ***link*** operand. Then, program execution continues at the address specified by ***dest***, sign extended to 21 bits, plus the current contents of the PC register. Table 5-2 shows the addressing calculations performed for the different flavors of **BAL**

Table 5-2. BAL Target Addressing Methodology

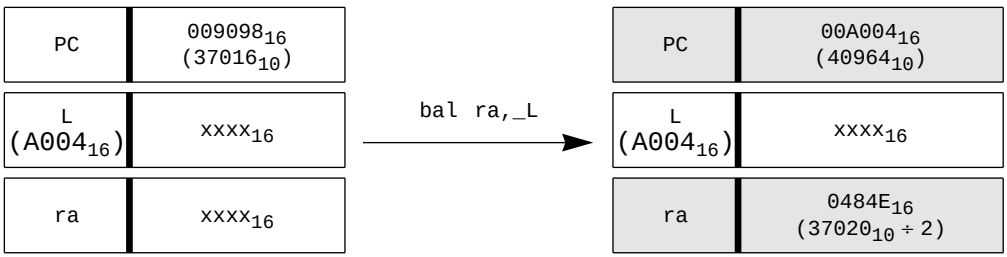
Displacement (signed)	Memory Model	Address Calculation
17 bits	Small	PC <- Truncate_17[a](PC + sext21[b](disp))
21 bits	Large	PC <- Truncate_21[c](PC + disp)

a. truncate_17: Truncate the address calculation result at 17 bits - 128K byte wrap around. Address bits 17 through 20, and address bit 0 are cleared to 0.
b. sext21: sign extend the displacement to 21 bits.
c. Truncate_21 means the whole calculation is performed at 21 bits, creating a 2 Mbyte wrap around. Address bit 0 is cleared to 0.

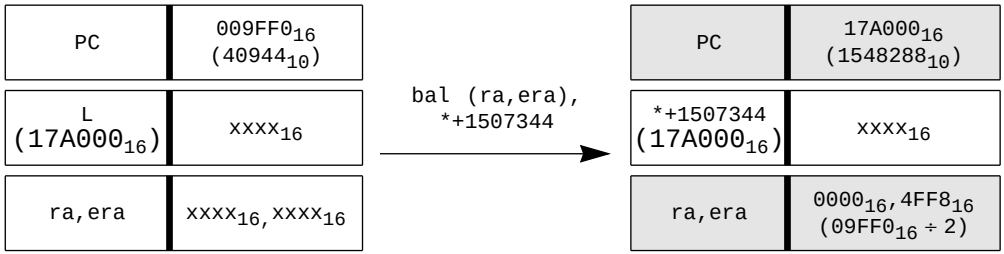
Flags: None

Traps: None

Examples: 1. Saves bits 1 through 16 of the PC register of the next sequential instruction in register RA, and passes execution control to the instruction labeled L by adding $00F6C_{16}$ to the current PC register.



2. Saves bits 1 through 16 of the PC register of the next sequential instruction in register ERA, bits 17 through 20 in register RA, and passes execution control to the instruction labeled **L** by adding 170010_{16} (1507344_{16}) to the current PC register.



BR

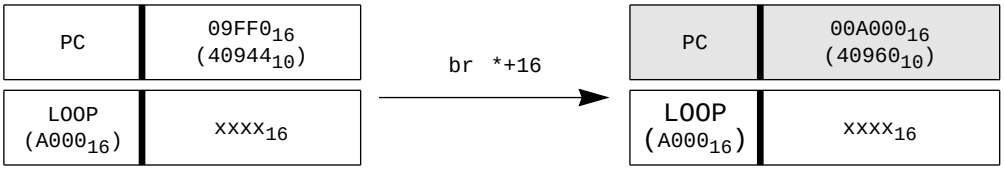
BR ***dest***
imm
disp

dest is sign extended to 21 bits and added to the current contents of the PC register. Displacement interpretation varies depending on compilation in small or large memory model, and the size of the displacement. The result is loaded into the PC register. Program execution continues at the location specified by the updated PC register.
Table "BR/BRcond Target Addressing Methodology" on page 5-9 describes the detailed address calculation performed for the **BR** instruction. For more information, refer to "BR/Bcond" on page 5-65.

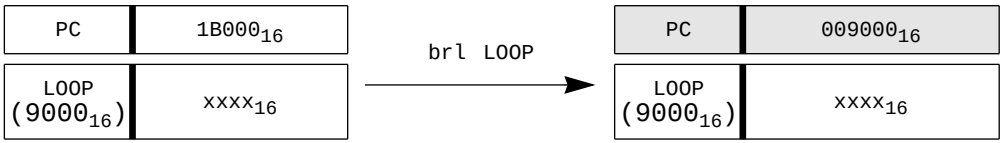
Flags: None

Traps: None

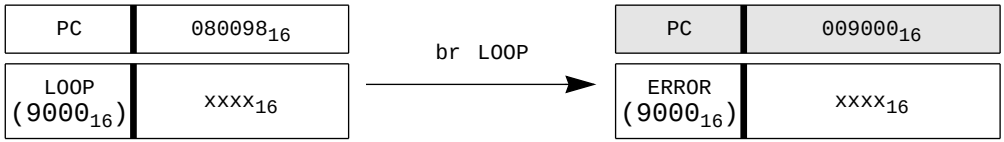
Examples: 1. Passes execution control to the instruction labeled **LOOP** by adding +16 to the PC register.



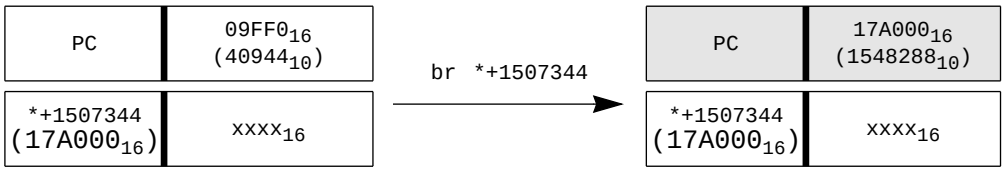
2. Small memory model: passes execution control to the instruction labeled **LOOP** by adding $E000_{16}$ to the PC register (wrap at 20000_{16}).



3. Large memory model: passes execution control to the instruction labeled **LOOP** by adding $188F68_{16}$ to the PC register (wrap at 200000_{16}).



4. Large memory model: passes execution control to a non-sequential instruction. The instruction passes execution control by adding 1507344 to the PC register.



For further examples, see the description of the **Bcond** instruction; **Branch** command executes the same as **Bcond**, assuming that the condition is always true.

CBITi Clear Bit in Integer

CBITB, CBITW

CBITi	***position***,	***dest***
	imm	abs/rel
	read.i	rmw.i

The **CBITi** instruction loads the ***dest*** operand from memory, clears the bit specified by ***position***, and stores it back into memory location ***dest***, in an uninterruptable manner. The ***position*** operand value must be in the range 0 to +7 if **CBITB** is used, and 0 to +15 if **CBITW** is used; otherwise, the result is unpredictable.

The addressing modes supported are:

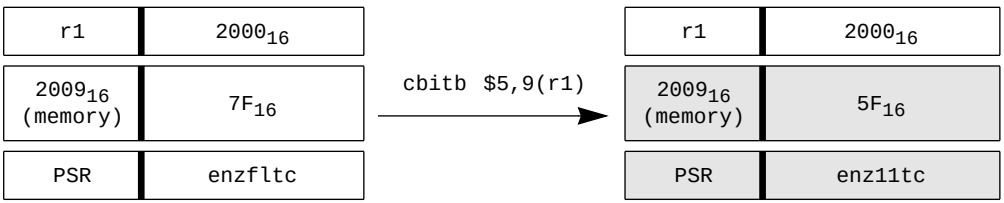
-rel: 0(Rn), where n is 0, 1, 8, or 9;
disp(Rn), where n is 0, 1, 8, or 9, and *disp* is 16-bit unsigned
-abs: 18-bit absolute address (covering the first 256 Kbyte addresses)

See "CBIT/SBIT/TBIT Addressing Methodology" on page 5-17.

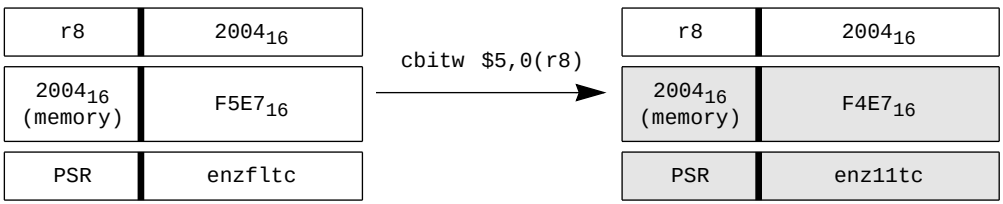
Flags: Before the specified bit is modified, its value is stored in PSR.F.

Traps: None

Examples: 1. Clears bit in position 5 in a byte operand in address 9 (R1).



2. Clears bit in position 8 in a word operand in address 0 (R8).



3. Clears bit in position 3 in a byte operand in address 30002_{16}

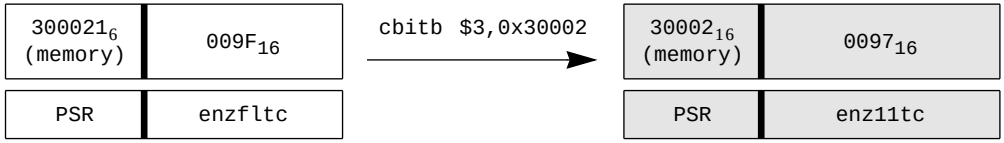


Table 5-3. CBIT/SBIT/TBIT Addressing Methodology

Addressing Format	Displacement Value Range	Allowed Address Range	Address Calculation
0(Rbase)	0	first 64K	DTruncate_18[a](zext21[b](Rbase))
disp16(Rbase)	0 - (64K − 1)	first 64K	DTruncate_18[a](zext21[b]Rbase + zext21[b](disp16))
abs18	0 - (256K-1)	first 256K	DTruncate_18[a](abs18)

a. DTruncate_18 means the whole calculation is performed at 18 bits, creating a 256 Kbyte wrap around.
b. zext21: zer extend the displacement to 21 bits (can be to 18 in this case).

CMPi — Compare Integer

CMPB, CMPW

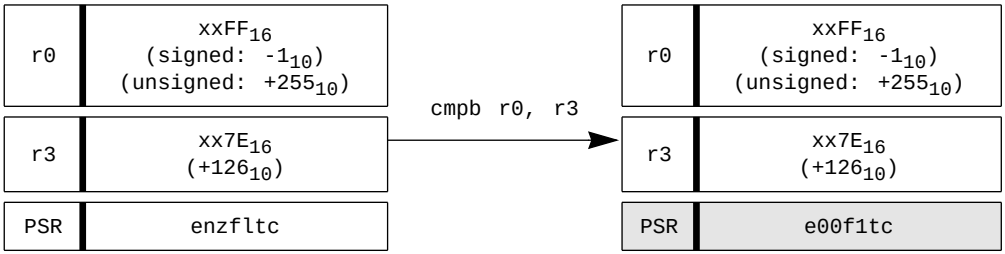
CMPi	***src1***,	***src2***
	reg/imm	reg
	read.i	read.i

The **CMPi** instruction subtracts the ***src1*** operand from the ***src2*** operand, and sets the PSR.Z, PSR.N, and PSR.L flags to indicate the comparison result. The PSR.N flag indicates the result of a signed integer comparison; the PSR.L flag indicates the result of an unsigned comparison. Both types of comparison are performed.

Flags: PSR.Z is set to 1 if ***src1*** equals ***src2***; otherwise it is cleared to 0. PSR.N is set to 1 if ***src1*** is greater than ***src2*** (signed comparison); otherwise it is cleared to 0. PSR.L is set to 1 if ***src1*** is greater than ***src2*** (unsigned comparison); otherwise it is cleared to 0.

Traps: None

Example: Compares low-order bytes in registers R0 and R3.



BCond0i

Compare Register to 0 and Conditional Branch

BEQ0B, BEQ0W, BNE0B, BNE0W

BCond0i	***src***,	***dest***
	reg,	imm
	read.i	disp

The **BCond0i** instruction compares the signed contents of ***src*** (registers 0, 1, 8, 9) to 0, and branches upon equality or non-equality (according to the ***cond***). The target address is determined by adding the 5-bit displacement (unsigned even 0-30) to the current value of the program counter. Only forward branching is supported.

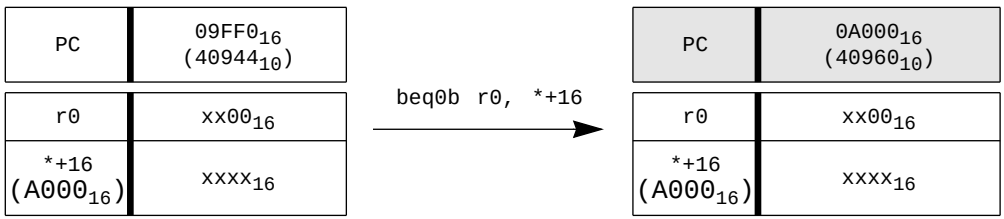
The instructions performs byte or word compares according to the ***i*** indicator.

Cond Code	Condition	True State
EQ	Equal	Rn is equal to 0
NE	Not Equal	Rn is not equal to 0

Flags: None

Traps: None

Example: Compares the low-order byte in register R0 to 0 and branches to the instruction labeled **L**, since the lower byte of R0 is 0, by adding 16 to the PC register.



BCond1i

Compare Register to 1 and Conditional Branch

BEQ1B, BEQ1W, BNE1B, BNE1W

Bcond1i	***src***,	***dest***
	reg,	imm
	read.i	disp

The **BCond1i** instruction compares the signed contents of ***src*** (registers 0,1,8,9) to 1, and branches on equality or non-equality (according to ***cond***). The target address is determined by adding the 5-bit displacement (unsigned) to the current value of the program counter. Only forward branching is supported.

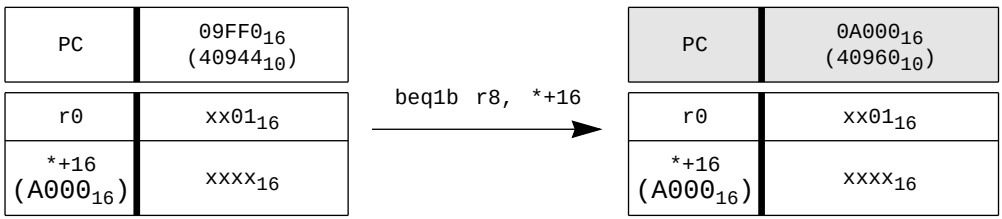
The instructions performs byte or word compares according to the ***i*** indicator.

Cond Code	Condition	True State
EQ	Equal	Rn is equal to 1
NE	Not Equal	Rn is not equal to 1

Flags: None

Traps: None

Example: Compares low-order byte in register R8 to 1, and branches to the instruction labeled **L**, since the lower byte of R0 is 1, by adding 16 to the PC register.



DI

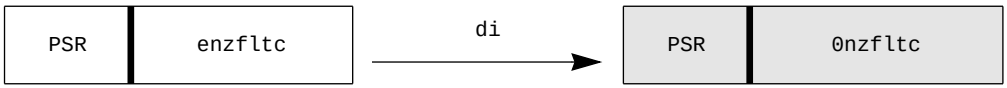
```
DI
```

The **DI** instruction clears PSR.E to 0. Maskable interrupts are disabled regardless of the value of PSR.I.

Flags: PSR.E is cleared to 0.

Traps: None

Example: Clears the PSR.E bit.



EI

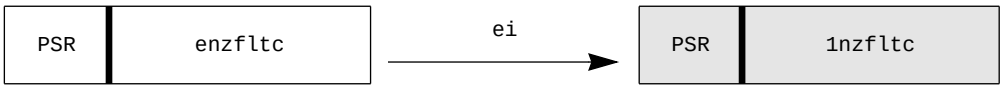
```
EI
```

The **EI** instruction sets PSR.E to 1. If PSR.I is also 1, maskable interrupts are enabled.

Flags: PSR.E is set to 1.

Traps: None

Example: Sets the PSR.E bit to 1.



EIWAIT

```
EIWAIT
```

The **EIWAIT** instruction suspends program execution until an interrupt occurs. This instruction also sets the PSR.E bit, enabling interrupt to occur. An interrupt restores program execution by passing it to an interrupt service procedure. When the **EIWAIT** instruction is interrupted, the return address saved on the stack is the address of the instruction following the **EIWAIT** instruction.

Flags: None

Traps: None

Example:

```
eiwait
```

EXCP

EXCP ***vector***

The **EXCP** instruction activates the trap specified by the ***vector*** operand. The return address pushed onto the interrupt stack is the address of the **EXCP** instruction itself. Specifying an **EXCP** with a reserved vector ***operand*** results in an Undefined (**UND**) exception.

Flags: None

Traps: The traps that occur are determined by the value of the ***vector*** operand, as shown in the following table.

Vector	Trap Name
SVC	Supervisor Call
DVZ	Division by Zero
FLG	Flag
BPT	Breakpoint
UND	Undefined Instruction
otherwise	*reserved*

Example: Activates the Supervisor Call Trap.

```
excp svc
```

JEQ, JNE, JCS, JCC, JHI, JLS, JGT, JLE, JFS, JFC, JLO, JHS, JLT, JGE

Jcond ***dest***
reg/regp
addr.W/L

If the condition specified by **cond** is true, the **Jcond** instruction causes a jump in program execution. Program execution continues at the address specified in the ***dest*** register (or register pair for large memory model), by loading the lower index register's contents into bits 1 through 16, and for large memory model, bits 0 through 3 of the higher index register into bits 17 through 20 of the PC register. Bits 0 (and 17 through 20 for small memory model) of the PC are cleared to 0. If the condition is false, execution continues with the next sequential instruction.

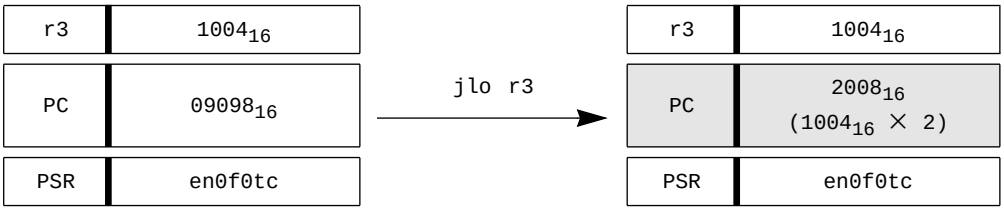
cond is a two-character condition code that describes the state of a flag or flags in the PSR. If the flag(s) are set as required by the specified **cond**, the condition is true; otherwise, the condition is false. The following table describes the possible **cond** codes and the related PSR flag settings:

cond Code	Condition	True State
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal	Z or N flag is 1

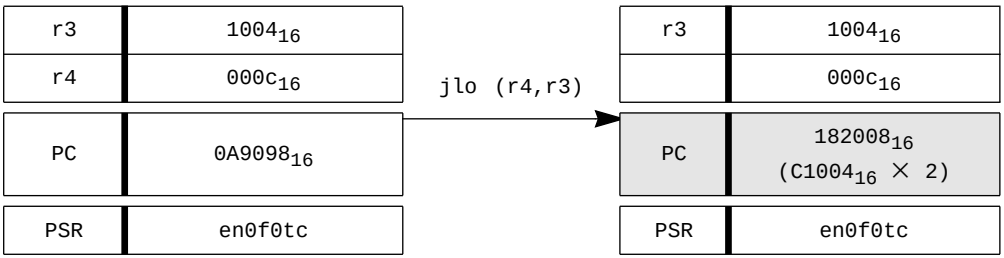
Flags: None

Traps: None

Examples: 1. Small memory model: loads the address held in R3 into bits 1 through 16 of the PC register, and program execution continues at that address, if the PSR.Z and PSR.L flags are 0.



2. Large memory model: loads the address held in R3, R4 into the bits 1 through 20 of the PC register. Program execution continues at that address, if the PSR.Z and PSR.L flags are 0.



JAL

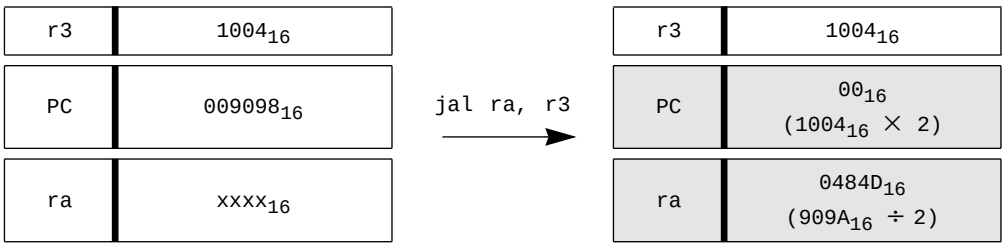
JAL	***link***,	***dest***
	reg/regp,	reg/regp
	write.W/L	addr.W/L

Program execution continues at the address specified in the ***dest*** register (or register-pair for large memory model), by loading the lower index register's contents into bits 1 through 16 of the PC register, and, for large memory model, by also loading bits 0 through 3 of the higher index register into bits 17 through 20 of the PC register. Bit 0, and for small memory model, bits 17-20 of the PC register are cleared to 0. Bits 1 through 20 of the address of the next sequential instruction are stored in the register, or register-pair, specified by the ***link*** operand; For small memory model, only bits 1 through 16 are stored in the single ***link*** register; For large memory model, bits 17 through 20 are also stored into bits 0 through 3 of the higher index ***link*** register.

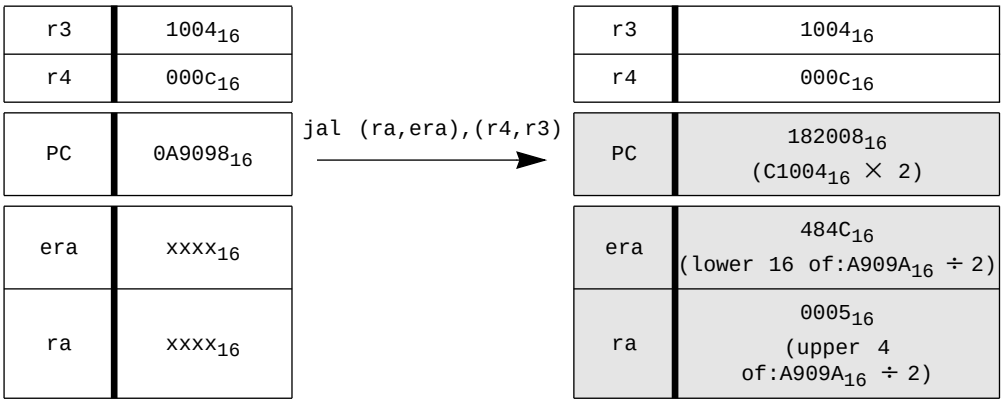
Flags: None

Traps: None

Examples: 1. Small memory model: loads the address held in R3 into the bits 1 through 16 of the PC register. Program execution continues at that address. Bits 1 through 16 of the address of the next sequential instruction are stored in register RA.



2. Large memory model: loads the address held in R3, R4 into the bits 1 through 20 of the PC register. Program execution continues at that address. Bits 1 through 16 of the address of the next sequential instruction are is stored in register ERA, and bits 17 through 20 of the address are stored in ERA+1.



JUMP

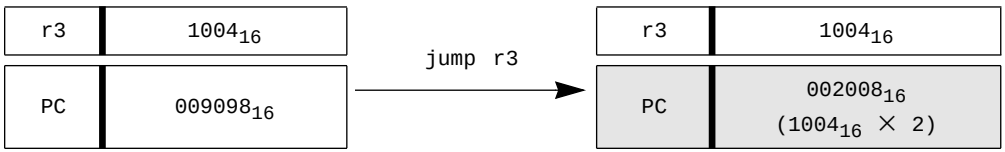
JUMP ***dest***
reg/regp
addr.W/L

Program execution continues at the address specified in the ***dest*** register (or register pair for large memory model), by loading the lower index register's contents into bits 1 through 16, and for large memory model, bits 0 through 3 of the higher index register into bits 17 through 20 of the PC register. Bits 0 (and 17 through 20 for small memory model) of the PC register are cleared to 0.

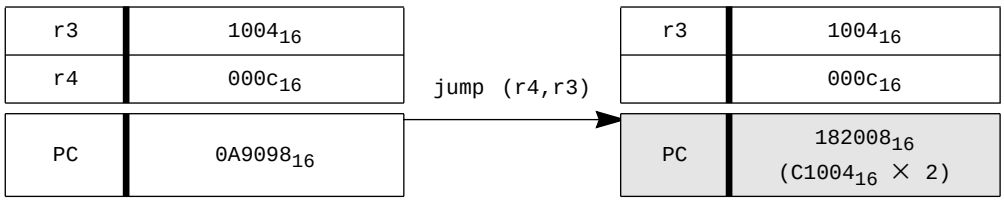
Flags: None

Traps: None

Examples: 1. Small memory model: loads the address held in R3 into bits 1 through 16 of the PC register. Program execution continues at that address.



2. Large memory model: loads the address held in R3, R4 into the bits 1 through 20 of the PC register. Program execution continues at that address.



LOADi Load from Memory

LOADB, LOADW

LOADi	***src***,	***dest***
	abs/rel/far	reg
	read.i	write.i

The **LOADi** instructions load the ***src*** operand from memory, and places it in the ***dest*** register operand.

For far execution, a register-pair is used as base, with bits 0-4 of the upper register translating to bits 16-20 of the address. Table 5-4 shows the addressing methodology for the instruction. For more information, and CR16A/CR16B inconsistencies, see “Load/Store far” on page 5-65.

Table 5-4. LOAD/STOR Memory Addressing Methodology

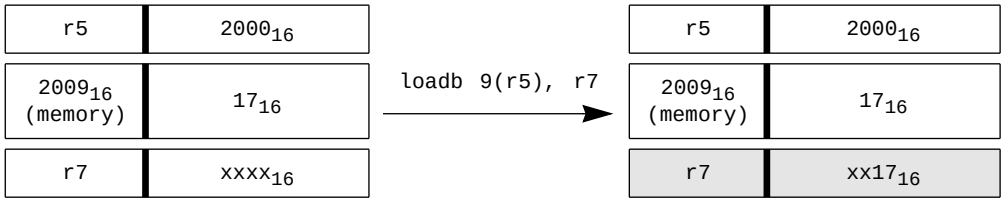
Addressing Modes	Displacement Range	Indexing Range (reg)	Instr. Len. Byte	Addr. Range	Address Calculation
disp5(Rbase)	0 to 31	64K	2	64K	Truncate_18[a] (zext21[b](Rbase) + zext21[b](disp))
disp16(Rbase)[c]	0 to 64K−1	64K	4	128K	Truncate_18[a] (zext21[b](Rbase) + sext21[d](disp))
disp18(Rbase)	−128K to 128K−1 or 0 to 256K−1	64K	4	256K	Truncate_18[a] (zext21[b](Rbase) + sext21[d](disp))
abs18	0 to 256K−1	---	4	256K	Truncate_18[a] (abs18)
disp18(Rbase+1, Rbase)	−128K to 128K−1	2M	4	2M	Truncate_21[e] (reg-pair + sext21[d](disp))

a. Truncate_18: Truncate the address calculation result at 18 bits - 256 Kbyte wrap around. Address bits 18 through 20 are cleared to 0.
b. zext21: zero extend to 21 bits.
c. Available only in ’STORi $imm4,disp(Rbase)’ instruction format.
d. sext21: sign extend the displacement to 21 bits.
e. Truncate_21 means the whole calculation is performed at 21 bits, creating a 2 Mbyte wrap around.

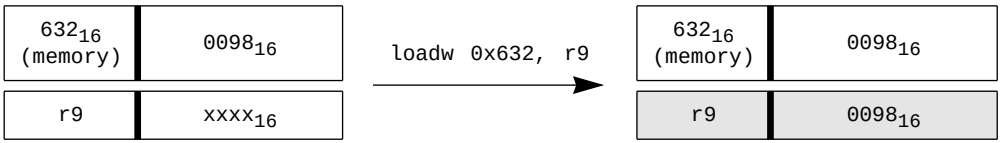
Flags: None

Traps: None

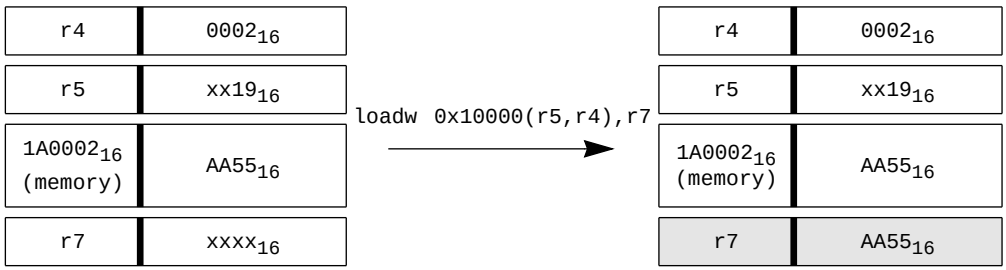
Examples: 1. Loads a byte operand in address 9 (R5) to the low-order byte of register R7. The remaining byte of register R7 is unaffected.



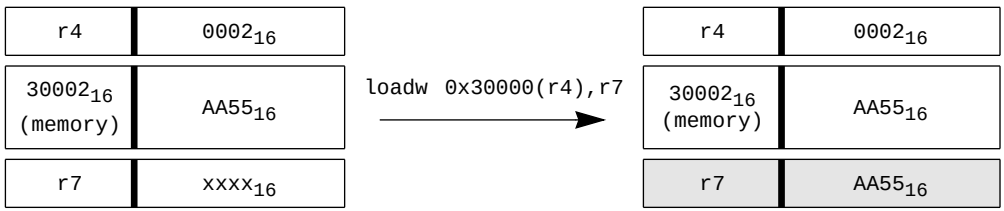
2. Loads a word operand in address 632 to register R9.



3. Loads a word operand in address $1A0002_{16}$ to register R7. The address is formed by adding 10000_{16} to the value in R4, concatenated with the value in R5.



4. Loads a word operand in address 30002_{16} to register R7. The address is formed by adding 30000_{16} to the value in R4.



LOADM Load Multiple Registers From Memory

LOADM

LOADM ***count***
imm
read,

The **LOADM** instruction loads up to four adjacent registers from memory. ***count*** reflects the total number registers to be loaded. (i.e., ***count*** = 1-4).

The instruction always operates on a fixed set of registers:

- r0 contains the address of the first word in memory to be loaded;
- r2 is loaded with the lowest address word;
- r3 through r5 are loaded from the next ***count***-1 consecutive addresses

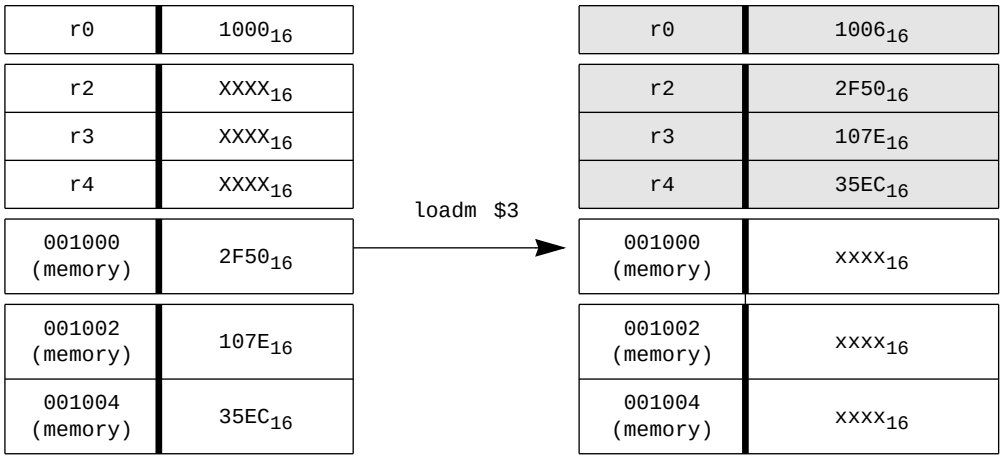
r0 is adjusted (incremented) by 2 for each word loaded, and therefore points to the next unread word in memory at the transfer-end.

This instruction is not interruptable.

Flags: None

Traps: None

Example: Loads three registers, from memory:



LPR

LPR	***src***,	***dest***
	reg	procreg
	read.W	write.W

The **LPR** instruction copies the ***src*** operand to the processor register specified by ***dest***.

If ***dest*** is ISP, only bits 0 through 15 are written, and the least significant bit (bit 0) and the five most significant bits (bits 16-20) of the address are cleared to 0. If ***dest*** is INTBASEL, bit 0 is cleared to 0. If ***dest*** is INTBASEH, bits 5 through 15 are always written as 0.

The following processor registers may be loaded:

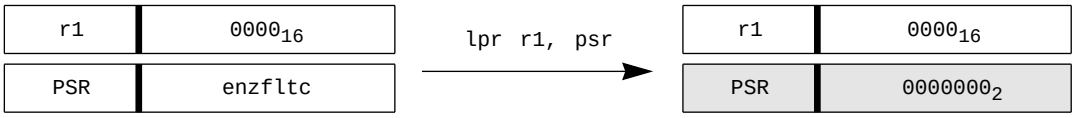
Register	procreg
Processor Status Register	PSR
Configuration Register	CFG
Interrupt Base Low Register	INTBASEL
Interrupt Base High Register	INTBASEH
Interrupt Stack Pointer	ISP
Debug Status Register	DSR
Debug Condition Register	DCR
Compare Address Register Low	CARL
Compare Address Register High	CARH

Refer to “REGISTER SET” on page 2-6 and to Chapter 4 “ADDITIONAL TOPICS” for more information on these registers.

Flags: PSR flags are affected by the values loaded into them. Otherwise, no PSR flags are affected.

Traps: None

Example: Loads register **PSR** from register R1.



LSHi	*count,*	*dest*
	reg/imm	reg
	read.B	write.i

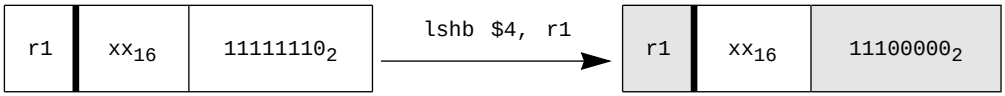
The **LSHi** instruction performs a logical shift on the ***dest*** operand as specified by the ***count*** operand.

The ***count*** operand is interpreted as a signed integer; the ***dest*** operand is interpreted as an unsigned integer. The sign of ***count*** determines the direction of the shift. A positive ***count*** specifies a left shift; a negative ***count*** specifies a right shift. The absolute value of ***count*** gives the number of bit positions to shift the ***dest*** operand. The ***count*** operand value must be in the range −7 to +7 if **LSHB** is used, and −15 to +15 if **LSHW** is used; otherwise, the result is unpredictable. All bits shifted out of ***dest*** are lost, and bit positions emptied by the shift are filled with zeros.

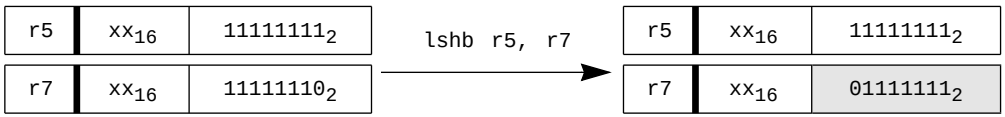
Flags: None

Traps: None

Examples: 1. Shifts the low-order byte of register R1 four bit positions to the left. The remaining bytes of register R1 is unaffected.



2. Reads a byte from register R5. Based on this value, it shifts the low-order byte of register R7. The remaining bytes of register R7 is unaffected.



MOVB, MOVW, MOVD

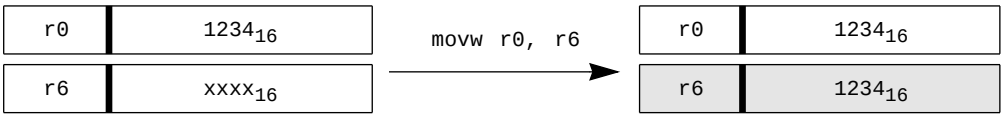
MOVi	***src***,	***dest***
	reg/imm	reg/regp
	read.i	write.i

The **MOVi** instructions copy the ***src*** operand to the ***dest*** register. For **MOVD**, only a 21-bit immediate ***src*** operand is supported, and the ***dest*** operand is a pair of adjacent registers. In this case, bits 5 through 15 of the higher-index register are always cleared to 0.

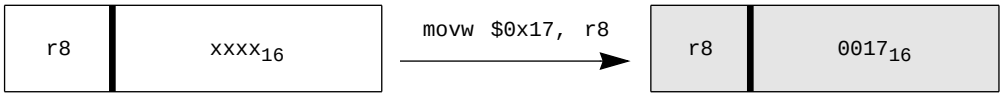
Flags: None

Traps: None

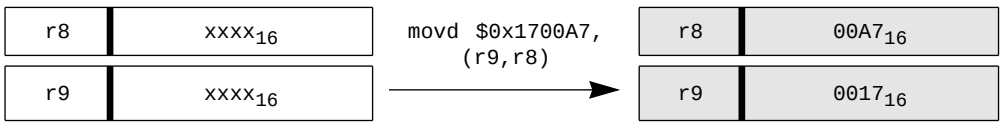
Examples: 1. Copies the contents of register R0 to register R6.



2. This example sets R8 to the value 17_{16}.



3. Sets register-pair R9, R8 to the value of $1700A7_{16}$.



MOVXB

MOVXB*src*, *dest*

reg	reg
read.i	write.W

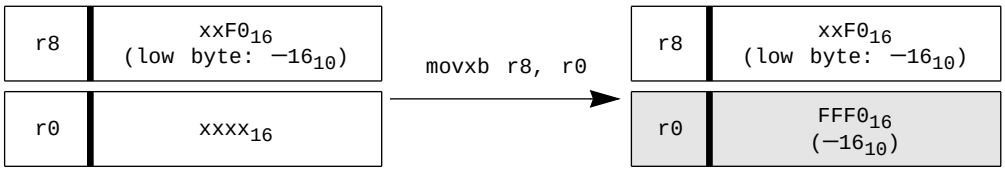
The **MOVXB** instruction converts the signed integer ***src*** operand to the word ***dest*** operand. The sign is preserved through sign-extension.

Flags: None

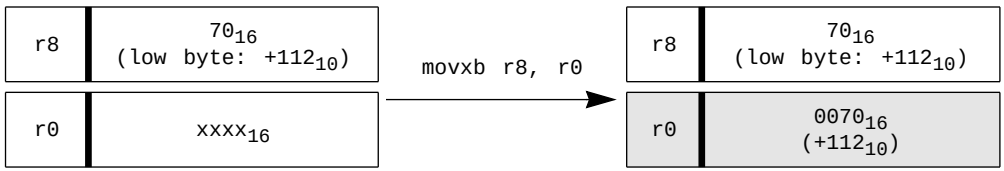
Traps: None

Examples: These examples copy the low-order byte of register R8 to the low-order byte of register R0, and extend the sign bit of the byte through the next 8 bits of register R0.

1. Illustrates negative sign extension.



2. Illustrates positive sign extension.



MOVZB

MOVZB*src*, d***est***

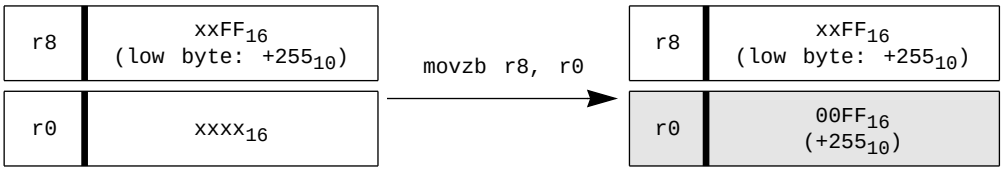
reg	reg
read.i	write.W

The **MOVZB** instruction converts the unsigned integer ***src*** operand to the unsigned word ***dest*** operand. The high-order bits are filled with zeros.

Flags: None

Traps: None

Example: Copies the low-order byte of register R8 to the low-order byte of register R0, and sets the next 8 bits of register R0 to zero.



MULB, MULW

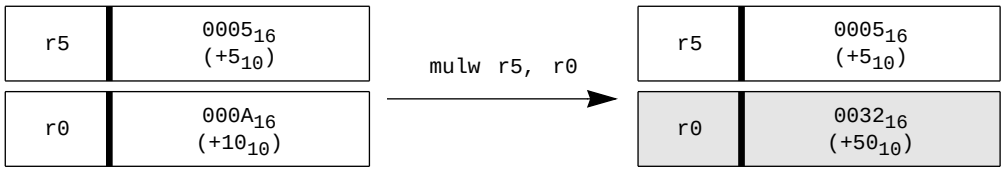
MULI	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **MULi** instructions multiply the ***src*** operand by the ***dest*** operand and places the result in the ***dest*** operand. Both operands are interpreted as signed integers. If the resulting product cannot be represented exactly in the ***dest*** operand, then the high-order bits are truncated.

Flags: None

Traps: None

Example: Multiplies register R5 by R0, and places the result in register R0.



MULSB Signed Multiply Byte, Word Result

MULSB

MULSW	***src***,	***dest***
	reg	reg
	read.B	rmw.W

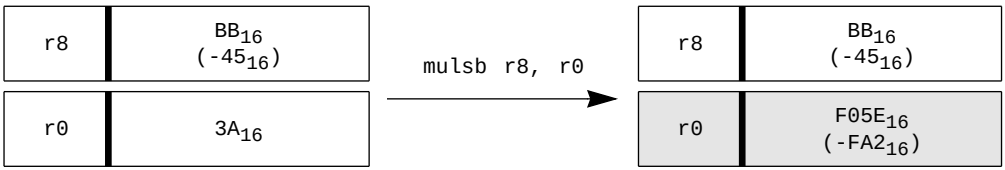
The **MULSB** instruction multiplies the 8-bit ***src*** operand by the 8-bit ***dest*** operand, and places the 16-bit result in the ***dest*** register.

Both source and destination operands are viewed as signed 8-bit operands, and the result is a signed 16-bit result.

Flags: None

Traps: None

Example: Multiplies signed register R8 by R0, and places the result in register R0.



MULSW

MULSW	***src***,	***dest***
	reg	regp
	read.W	rmw.D

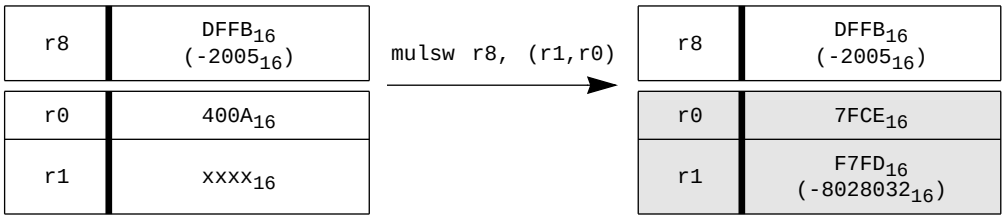
The **MULSW** instruction multiplies the 16-bit ***src*** operand by the 16-bit ***dest*** operand, and places the 32-bit result in the register-pair ***dest+1***, ***dest***.

Both source and destination operands are viewed as signed 16-bit operands, and the result is a signed 32-bit result.

Flags: None

Traps: None

Example: Multiplies signed register R8 by R0, and places the result in registers R1, R0.



MULUW Unsigned Multiply Word Long Result

MULUW

MULUW	***src***,	***dest***
	reg	regp
	read.W	rmw.D

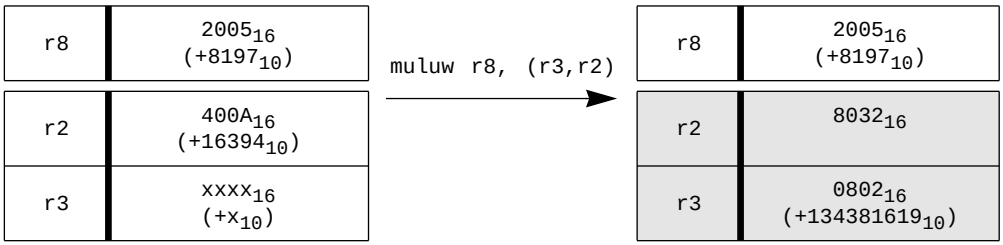
The **MULUW** instruction multiplies the 16-bit ***src*** operand by the 16-bit ***dest*** operand, and places the 32-bit result in the register-pair ***dest+1***,***dest***. In this instruction, the choice of source registers for the first operand are restricted to R0, R1, R8, or R9.

Both source and destination operands are viewed as unsigned 16-bit operands, and the result is an unsigned 32-bit result.

Flags: None

Traps: None

Example: Multiplies unsigned register R8 by R2, and places the result in registers R3, R2.



NOP

```
NOP
```

The **NOP** instruction passes control to the next sequential instruction. No operation is performed.

Flags: None

Traps: None

Example:

```
nop
```

ORB, ORW

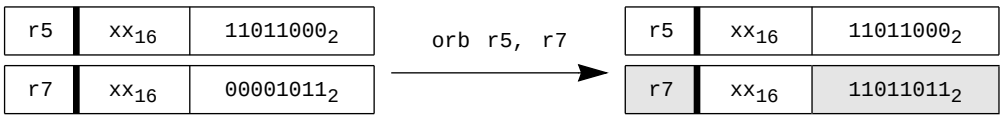
ORi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **ORi** instructions perform a bitwise logical OR operation on the ***src*** and ***dest*** operands, and places the result in the ***dest*** operand.

Flags: None

Traps: None

Example: ORs the low-order bytes of registers R5 and R7, and places the result in the low-order byte of register R7. The remaining byte of register R7 is unaffected.



POPrt Pop Multiple Registers From Stack

POP, POPRET

POPrt	***count***,	***src***
	imm,	reg
	read,	write.W

The **POPrt** instructions restore up to four adjacent registers from the user stack. The registers are be defined by ***src*** register, with up to three more adjacent registers. (e.g., ***src***, ***src***+1, ***src***+2, ***src***+3). ***count*** reflects the total number registers to restore. (i.e., ***count*** = 1-4). ***src*** is loaded with the value residing at the lowest address (top of stack). The stack pointer (**SP**) is adjusted (incremented) accordingly.

This instruction is not interruptable.

Note: In no way should the parameters to **POPrt** indicate a restore of registers r15 (**SP**) and beyond. Such parameters (i.e., count/src pairs of 1-4/**SP, 2-4/ra, 3-4/era, 4/r12)** lead to unpredictable results.

After the pop operation has ended, the processor can return control to a calling routine, according to the **rt** switch in the instruction.

rt Switch	**Operation**	**Implied Instruction**
Not specified	No-return	--
RET	Return (small memory model)	**JUMP** ra
	Return-Long (large memory model	**JUMP** (ra,era)

The **POPrt** instructions do not change the contents of memory locations indicated by an asterisk *. However, information that is outside the stack should be considered unpredictable for other reasons.

Flags: None

Traps: None

Examples: 1. Pops three registers, starting with R3, from the stack:

PC	$00F034_{16}$			PC	$00F036_{16}$
SP	1000_{16}			SP	1006_{16}
r3	$XXXX_{16}$			r3	$2F50_{16}$
r4	$XXXX_{16}$			r4	$107E_{16}$
r5	$XXXX_{16}$	pop 3,R3 →		r5	$35EC_{16}$
001000 (stack)	$2F50_{16}$			001000 (stack)	$xxxx_{16}$ *
001002 (stack)	$107E_{16}$			001002 (stack)	$xxxx_{16}$ *
001004 (stack)	$35EC_{16}$			001004 (stack)	$xxxx_{16}$ *

2. Pops four registers from the stack, and executes a **JUMP** (**RA**,**ERA**) (Large-memory-model):

PC	$00F034_{16}$			PC	$066BD8_{16}$ ({ra,era) x 2}
SP	1000_{16}			SP	1008_{16}
r11	$XXXX_{16}$			r11	$2F50_{16}$
r12	$XXXX_{16}$			r12	$107E_{16}$
ra	$XXXX_{16}$			ra	$35EC_{16}$
era	$XXXX_{16}$	popret 4,R11 →		era	0003_{16}
001000 (stack)	$2F50_{16}$			001000 (stack)	$xxxx_{16}$ *
001002 (stack)	$107E_{16}$			001002 (stack)	$xxxx_{16}$ *
001004 (stack)	$35EC_{16}$			001004 (stack)	$xxxx_{16}$ *
001006 (stack)	0003_{16}			001006 (stack)	$xxxx_{16}$ *

PUSH

PUSH	count	***src***
	imm	reg
	read	read.W

The **PUSH** instruction saves up to four adjacent registers, onto the user stack, with the ***count*** parameter denoting how many registers are saved (i.e., ***count*** = 1-4). The registers are defined by the ***src*** register, with up to three more adjacent registers. (e.g., ***src***, ***src***+1, ***src***+2, ***src***+3).

The registers are stored in ***count*** consecutive words on the stack, with ***src*** contents residing at the lowest address (top of stack).

The stack pointer is adjusted (decremented) accordingly, and points to the top of stack at the end of the instruction. This instruction is not interruptable.

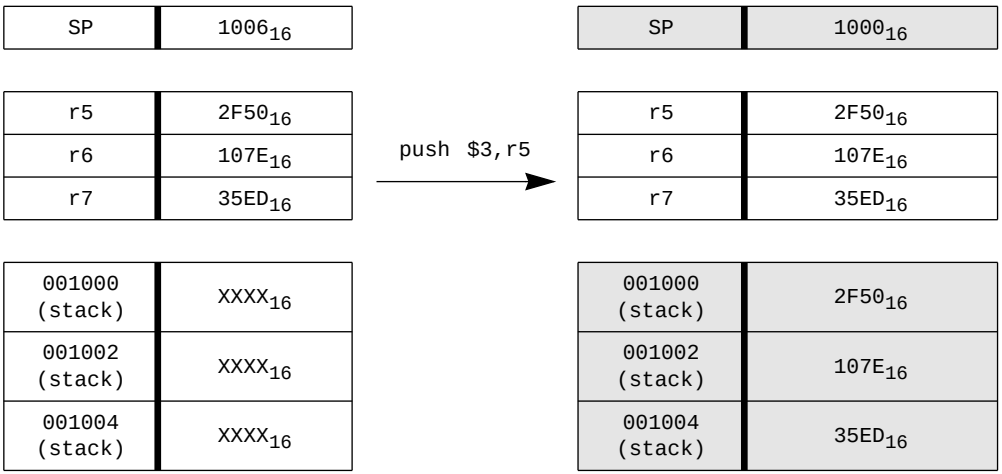
Register pairs containing Long addresses (21 bits), stored within a single **PUSH** instruction, would comply with "Little-endien" methodology.

Note: In no way should the parameters to **PUSH** indicate a save of registers r15 (**SP**) and beyond. Such parameters (i.e., count/src pairs of 1-4/**SP**, 2-4/**ra**, 3-4/**era**, 4/**r12)** lead to unpredictable results.

Flags: None

Traps: None

Example: Pushes three registers, starting with R5, on the stack:



RETX

RETX

The **RETX** instruction returns control from a trap service procedure. The following steps are performed:

1. The instruction pops a 16-bit return address from the interrupt stack, and loads it into bits 1 through 16 of the PC register.

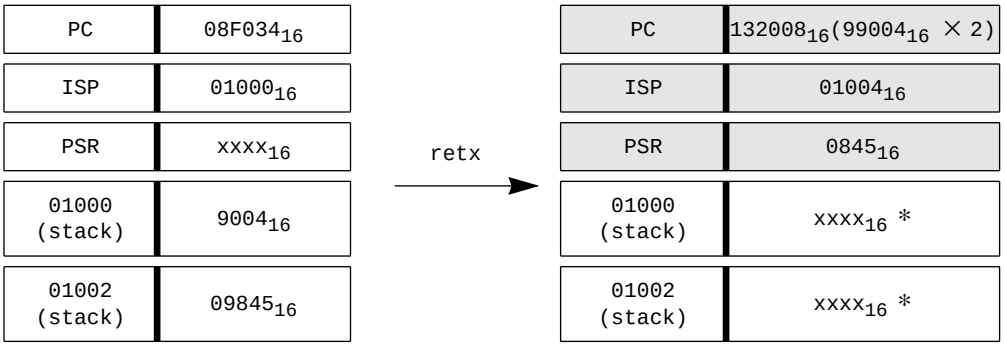
2. The instruction then pops a 16-bit value containing the upper four return address bits, and the 12-bit PSR value from the interrupt stack, storing the four address bits into bits 17 through 20 of the PC register, and the 12 PSR bits into the PSR register.

The **RETX** instruction does not change the contents of memory locations indicated by an asterisk (*). However, information that is outside the stack should be considered unpredictable for other reasons.

Flags: All PSR flag states are restored from the stack.

Traps: None

Example: Returns control from an interrupt service procedure.



SBITi — Set Bit

SBITB, SBITW

SBITi	***position***,	***dest***
	imm	abs/rel
	read.i	read+write.i

The **SBITi** instructions load the ***dest*** operand from memory, set the bit position specified by ***position***, and store it back into memory location ***dest***, in an uninterruptable manner. The ***position*** operand value must be in the range 0 to +7 if **SBITB** is used, and 0 to +15 if **SBITW** is used; otherwise, the result is unpredictable.

The memory modes supported are:

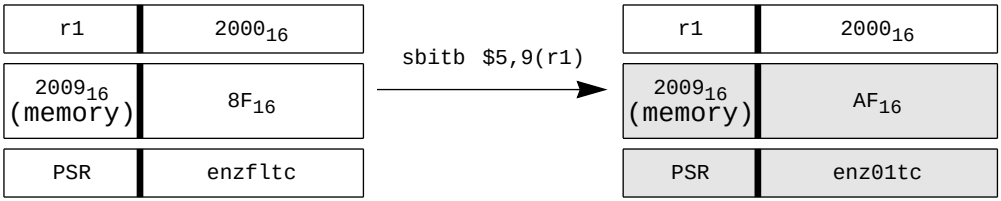
-*Rel*: 0(Rn), where n is 0, 1, 8, or 9;
disp(Rn), where n is 0, 1, 8, or 9, and *disp* is 16 bit unsigned
-*Abs*: 18-bit absolute address (covering the first 256K addresses)

See "CBIT/SBIT/TBIT Addressing Methodology" on page 5-17.

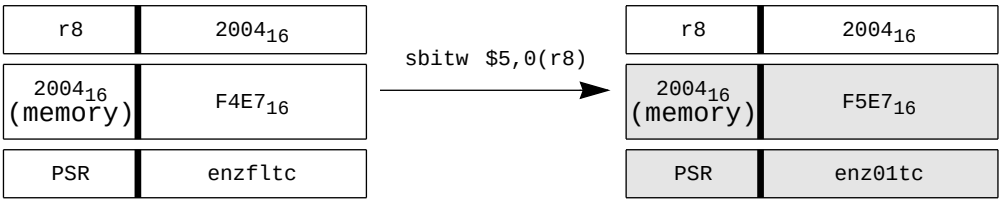
Flags: Before the specified bit is modified, its value is stored in PSR.F.

Traps: None

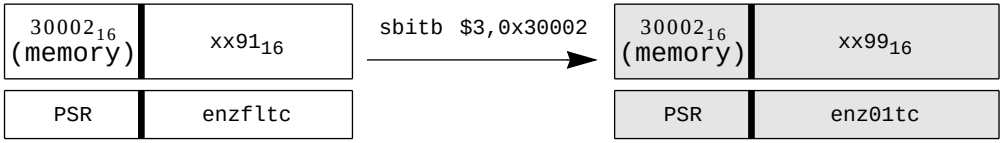
Examples: 1. Sets bit in position 5 in a byte operand in address 9 (R1).



2. Sets bit in position 8 in a word operand in address 0 (R8).



3. Sets bit in position 3 in a byte operand in address 30002_{16}



Scond — Save Condition as Boolean

SEQ, SNE, SCS, SCC, SHI, SLS, SGT, SLE, SFS, SFC, SLO, SHS, SLT, SGE

Scond ***dest***
reg
write.W

The **Scond** instruction sets the ***dest*** operand to the integer value 1 if the condition specified in **cond** is true, and clears it to 0 if it is false.

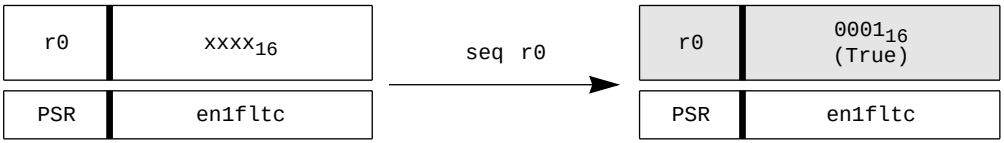
cond is a two-character condition code that specifies the state of a flag or flags in the PSR register. If the flag(s) are set as required by the specified **cond**, the condition is true; otherwise, the condition is false. The following table describes the possible **cond** codes and the related PSR flag settings:

cond Code	Condition	True State
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal	Z or N flag is 1

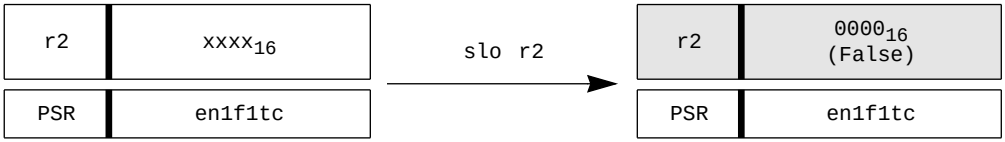
Flags: None

Traps: None

Examples: 1. Sets register R0 to 1 if the PSR.Z flag is set and to 0 if it is clear.



2. Sets register R2 to 1 if the PSR.Z and PSR.L flags are cleared and to 0 if they are not cleared.



SPR

SPR	*src,*	*dest*
	procreg	reg
	read.W	write.W

The **SPR** instruction stores the processor register specified by ***src***, in the ***dest*** operand. If ***src*** is ISP, only bits 0 through 15 of ***src*** are stored.

The following processor registers may be stored:

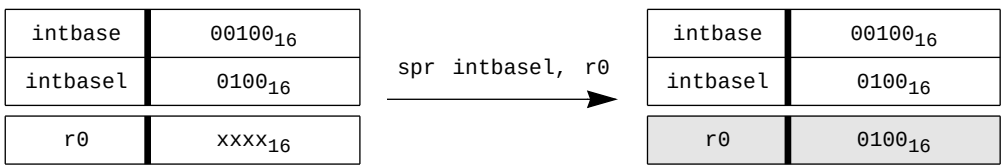
Register	procreg
Processor Status Register	PSR
Configuration Register	CFG
Interrupt Base Low Register	INTBASEL
Interrupt Base High Register	INTBASEH
Interrupt Stack Pointer	ISP
Debug Status Register	DSR
Debug Condition Register	DCR
Compare Address Register Low	CARL
Compare Address Register High	CARH

See “REGISTER SET” on page 2-6 for more information.

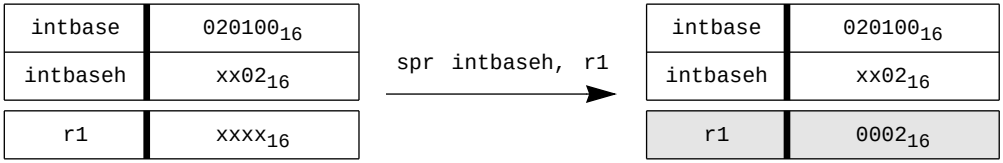
Flags: None

Traps: None

Examples: 1. Copies the INTBASEL register to register R0.



2. Copies the INTBASEH register to register R1.



STORi	***src***,	***dest***
	reg	abs/rel/far
	read.i	write.i
STORi	***src***,	***dest***
	imm4	abs/rel
	read.i	write.i

The **STORi** instructions store the ***src*** register operand in the ***dest*** memory operand.

For far execution, a register-pair is used as base, with bits 0-4 of the upper register translating to bits 16-20 of the address.

A second format of the instructions allow storing of a 4-bit immediate value into the ***dest*** memory operand. This format has limited addressing modes. The memory modes supported are:

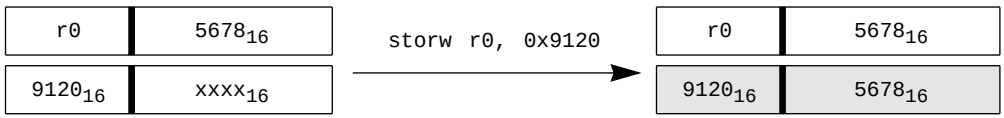
-*Rel*: 0(Rn), where n is 0, 1, 8, or 9;
disp(Rn), where n is 0, 1, 8, or 9, and *disp* is 16-bit unsigned
-*Abs*: 18-bit absolute address (covering the first 256K addresses)

Table 5-4 describes the addressing methodology for the instruction. See "Load/Store far" on page 5-65.

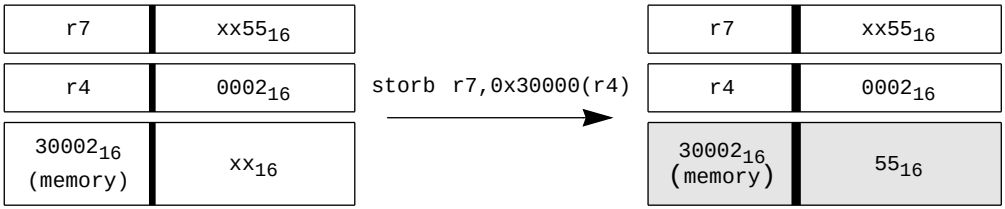
Flags: None

Traps: None

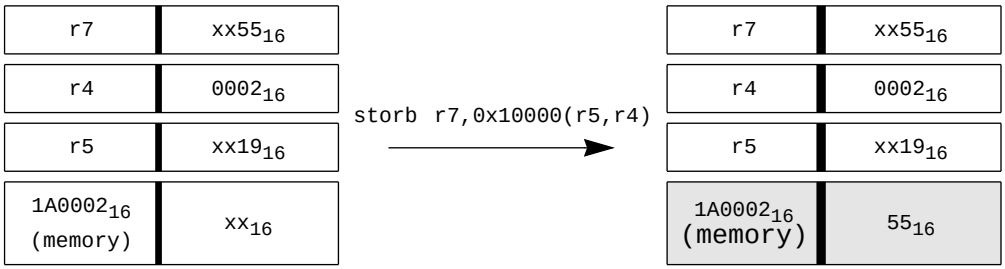
Examples: 1. Copies the contents of register R0 to the word at address 9120_{16}.



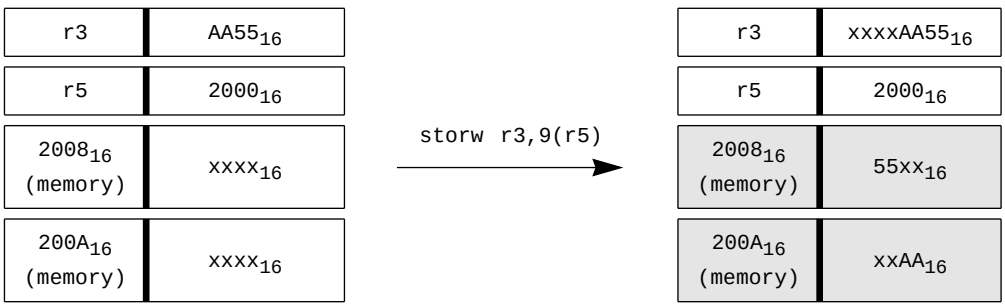
2. Stores the low-order byte from R7 at address 30002_{16}. The address is formed by adding 30000_{16} to the value in R4.



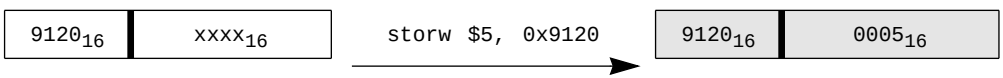
3. Stores the low-order byte from R7 at address $1A0002_{16}$. The address is formed by adding 10000_{16} to the value in R4 concatenated with the value in R5.



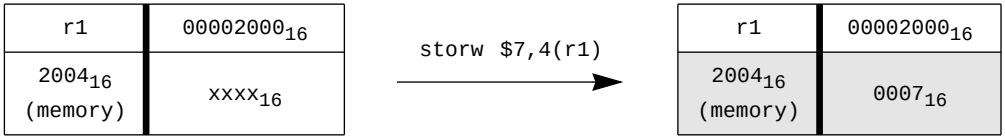
4. Copies the contents of register R3 to the non-aligned word at address 9 (R5).



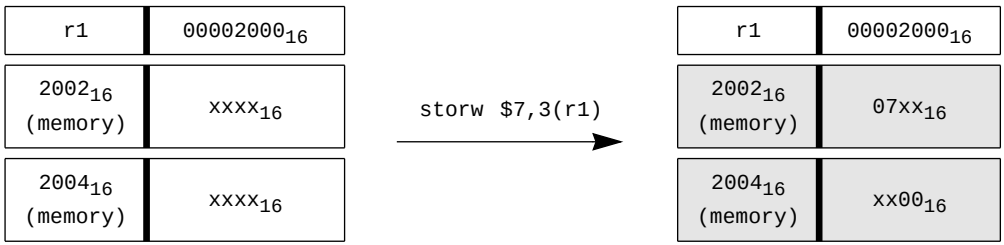
5. Stores the immediate value of 0x5 to the word at address 9120_{16}.



6. Stores the immediate value of 0x7 to the word at address 4 (R1).



7. Stores the immediate value of 0x7 to the non-aligned word at address 3 (R1).



STORM Store Multiple Registers To Memory

STORM

STORM ***count***
imm
read,

The **STORM** instruction stores up to four adjacent registers to memory. ***count*** reflects the total number registers to be stored. (i.e., ***count*** = 1-4).

The instruction always operates on a fixed set of registers:

- r1 contains the target address of the first word in memory;
- r2 is stored into the lowest address word;
- r3 through r5 are stored to the next ***count***-1 consecutive addresses

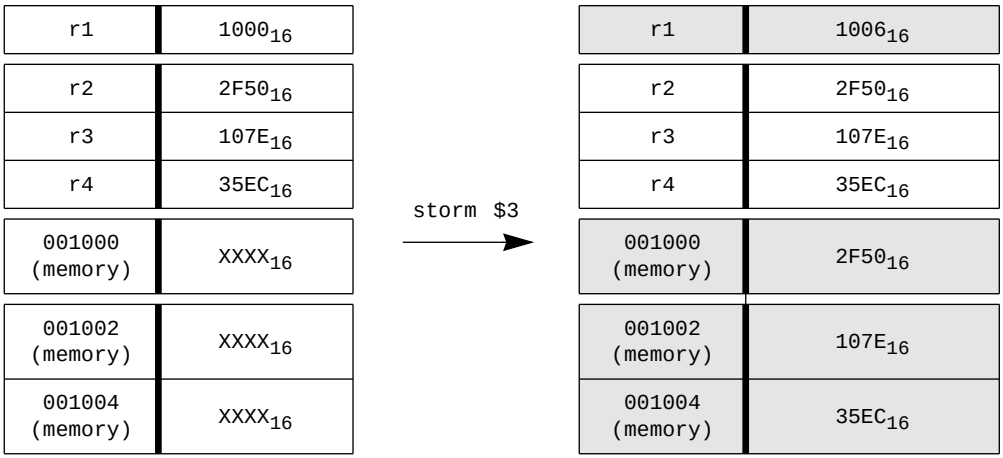
r1 is adjusted (incremented) by 2 for each word stored, and therefore points to the next unwritten word in memory at the transfer-end.

This instruction is not interruptable.

Flags: None

Traps: None

Example: Stores three registers into memory:



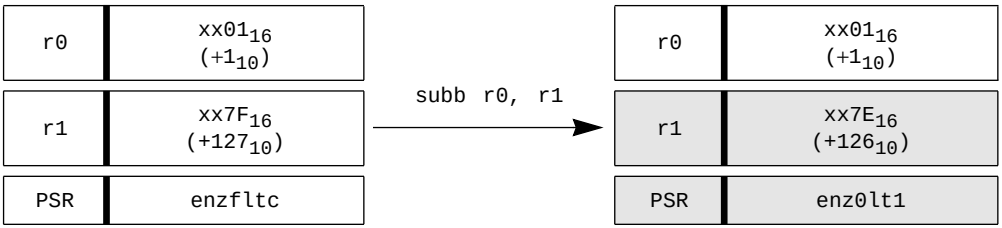
SUBi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **SUBi** instructions subtract the ***src*** operand from the ***dest*** operand, and places the result in the ***dest*** operand.

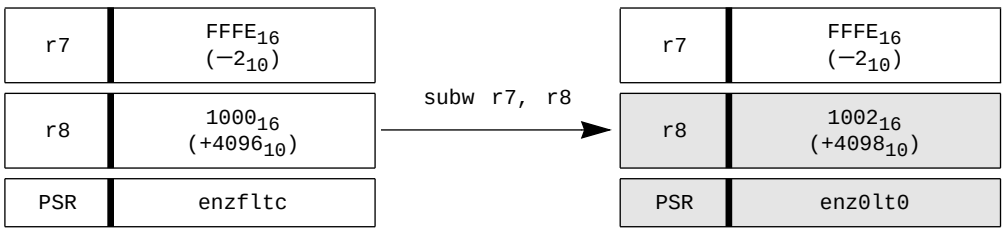
Flags: During execution of a **SUBi** instruction, PSR.C is set to 1 if a borrow occurs, and cleared to 0 if no borrow occurs. PSR.F is set to 1 if an overflow occurs, and cleared to 0 if there is no overflow.

Traps: None

Examples: 1. Subtracts the low-order byte of register R0 from the low-order byte of register R1, and places the result in the low-order byte of register R1. The remaining byte of register R1 is not affected.



2. Subtracts the word in register R7 from the word in register R8, and places the result in register R8.



SUBCi — Subtract with Carry

SUBCB, SUBCW

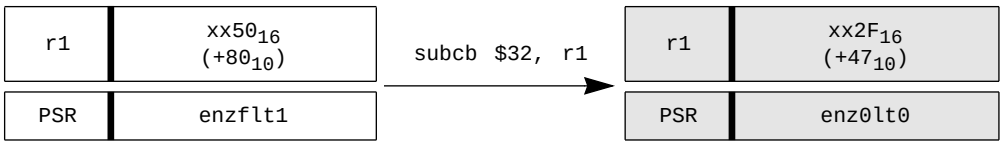
SUBCi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **SUBCi** instructions subtract the sum of the ***src*** operand and the PSR.C flag from the ***dest*** operand, and places the result in the ***dest*** operand.

Flags: PSR.C is set to 1 if a borrow occurs and cleared to 0 if there is no borrow. 0 PSR.F is set to 1 if an overflow occurs and cleared to 0 if there is no overflow.

Traps: None

Example: Subtracts the sum of 32 and the PSR.C flag value from the low-order byte of register R1 and places the result in the low-order byte of register R1. The remaining byte of register R1 is not affected.



TBIT, TBITB, TBITW

TBIT	***position,***	***src***
	reg/imm	reg
	read.W	read.W
TBITi	***position,***	***src***
	imm	abs/rel
	read.i	read.i

The **TBIT** instruction copies the bit located in register or memory location ***src*** at the bit position specified by ***position***, to the PSR.F flag. The memory-direct format of the instruction supports byte and word operations (**TBITB**, **TBITW**), while the register-sourced format does not. The ***offset*** value must be in the range 0 through 15 for a word operand, and in the range of 0 through 7 for a byte operand; otherwise, the result is unpredictable.

The memory-direct format supports only limited addressing modes:

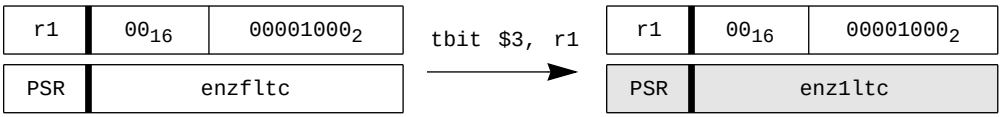
-*Rel*: 0(Rn), where n is 0, 1, 8, or 9;
disp(Rn), where n is 0, 1, 8, or 9, and *disp* is 16-bit unsigned
-*Abs*: 18 bit absolute address (covering the first 256K addresses)

See "CBIT/SBIT/TBIT Addressing Methodology" on page 5-17.

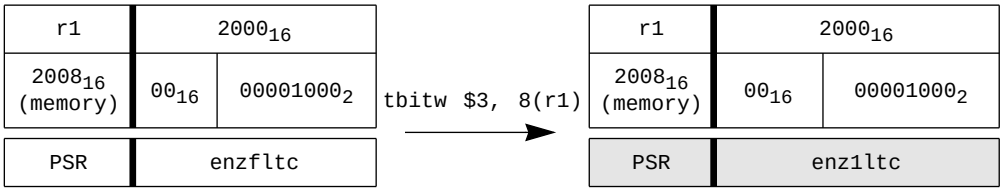
Flags: PSR.F is set to the value of the specified bit.

Traps: None

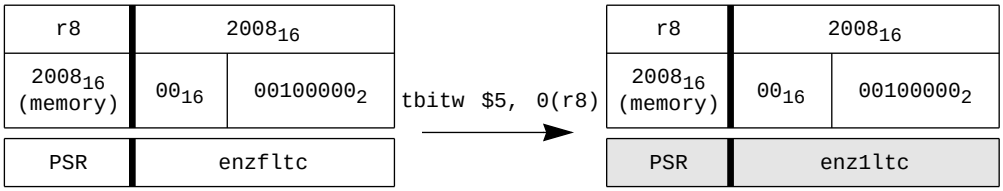
Examples: 1. Copies bit number 3, i.e., the fourth bit from the right, in register R1 to the PSR.F flag.



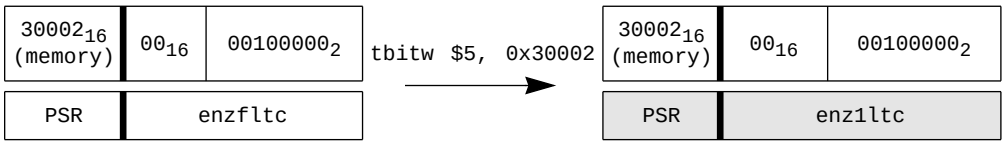
2. Copies bit number 3, i.e., the fourth bit from the right, in memory location 8 (R1) to the PSR.F flag.



3. Copies bit number 5, i.e., the sixth bit from the right, in memory location 0 (R8) to the PSR.F flag.



4. Copies bit number 5, i.e., the sixth bit from the right, in memory location 30002_{16} to the PSR.F flag.



WAIT

```
WAIT
```

The **WAIT** instruction suspends program execution until an interrupt occurs. An interrupt restores program execution by passing it to an interrupt service procedure. When the **WAIT** instruction is interrupted, the return address saved on the stack is the address of the instruction following the **WAIT** instruction.

Flags: None

Traps: None

Example:

```
wait
```

XORi — Bitwise Logical Exclusive OR

XORB, XORW

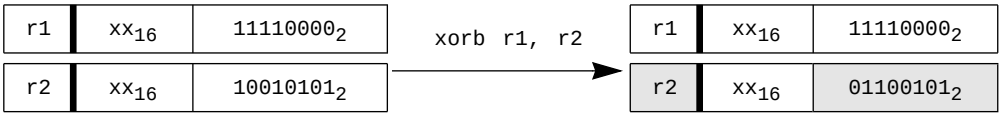
XORi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **XORi** instructions perform a bitwise logical exclusive OR operation on the ***src*** and ***dest*** operands, and places the result in the ***dest*** operand.

Flags: None

Traps: None

Example: XORs the low-order bytes of registers R1 and R2, and places the result in the low-order byte of register R2. The remaining byte of R2 is unaffected.



5.3 CR16B/CR16A INSTRUCTION INCOMPATIBILITIES

The CR16B differs from the CR16A by having 21 bits of address space for both code and data usage, versus the 17 bits of instruction address, and 18 bits of data space, available on the CR16A. These differences are usually transparent in most CR16A instructions, when executed on the CR16B. The two main exceptions are the **BR**/**Bcond** and the **LOAD**/**STORE** far instructions.

BR/Bcond This instruction manifests two characteristics in the CR16B, with relation to the displacement size.

For a medium displacement, when used in the small memory model (17-bit effective), the instruction still conforms to the CR16A's model: the displacement is sign-extended to 18 bits, added to the lower 18 bits of the PC register, and the result is truncated to 17 bits, resulting in a wrap-around at the 128 Kbyte boundary. This instruction only works in the first 128K of memory.

For a short displacement (9 bits), the new CR16B large memory model is used; the displacement is sign extended to 21 bits, and the whole 21 bits of the PC register are used for the address calculation (truncated at 21 bits), resulting in a wrap around at the 2 Mbyte boundary. This allows usage of the short format of the branch in side routines and program portions residing over the 128K boundary, thus keeping the code size small, at the expense of not conforming to the CR16A model. This only affects programs using the short branch instruction to reach the other end of the memory - not a recommended method by itself, and not used by the compilers.

For a medium displacement, when used in the large memory model, (21-bits), the whole 21 bits of the PC register are used for the address calculation (truncated at 21 bits), resulting in a wrap around at the 2 Mbyte boundary. This allows covering the whole 2 Mbyte instruction space of the CR16B, using the wrap-at-2M to reach the beginning of memory with just a 21 bit displacement (either signed or unsigned).

Load/Store far The **load**/**store** *far* instructions have not changed format in the CR16B, but since the address space has grown, there are two areas in which the CR16A and the CR16B differ:

1. The processor now uses bits 2-5 of the upper base address register-pair, as well as bits 0-1, to calculate the load-store address. If the program modified those bits to some non-zero value, a CR16B running this CR16A code would not perform correctly, whereas the CR16A would, since it disregards bits 2 and above. You should leave the upper bits of an address register-pair as 0, and not re-use them for other purposes.

2. Since the *far* format must cover the whole 2 Mbyte range, and still allow a signed displacement, the displacement is limited to −128K to +128K bytes only, to be sign extended for calculation. Using the 18-bit displacement as a base in the range of 128K to 256K now gives erroneous results.

Appendix A
INSTRUCTION EXECUTION TIMING

This appendix describes the factors which affect instruction execution timing in the CR16B. A CR16B based microprocessor may include a write buffer, and a Bus Interface Unit (BIU). This appendix does not describe instruction execution timing that depends on the architecture of such modules.

A.1 TIMING PRINCIPLES

Timing glossary

Clock Cycle - The unit of time for the clock tick. At 33 MHz operation, there are 33 million clock cycles per second.

Instruction Latency - The number of clock cycles required to process a given instruction, from the time it enters the pipeline, until it leaves the pipeline.

Instruction Throughput - The number of instructions that completed the execution stage in a given number of clock cycles.

Program Execution Time - The number of clock cycles that elapsed, from the time the first instruction began, until the last instruction left the pipeline. The program execution time depends on the instruction latency for each instruction in the program and the instruction throughput.

Timing factors

Under optimal conditions, the CR16B performs one instruction per clock cycle. At 33 MHz, this translates to 33 MIPS (Million Instructions Per Second). However, under a typical workload, unavoidable delays are caused by the pipeline and memory.

Memory access time

Each access to memory, when there are zero wait states, takes one clock cycle. This means that the data arrives one clock cycle after the address was issued. The access time for off-chip memory depends on the speed and configuration of the off-chip memory.

When an instruction fetch, a load or a store instruction accesses external memory, additional clock cycles may be added depending on the configuration of the CPU, i.e., the existence and depth of a write buffer, and the speed and configuration of the off-chip memory.

Evaluating execution time

To calculate the total program execution time, in clock cycles, combine the following:

1. The number of clock cycles required to execute each instruction.

2. Delays, in clock cycles, caused by contention for memory and stalled execution of instructions in the pipeline.

3. The number of clock cycles required to handle exceptional conditions, such as interrupts.

A.2 THE PIPELINE

Every instruction executed by the CR16B passes through the three stages in the pipeline.

- Instruction Fetch (IF) - The instruction is fetched from memory.
- Instruction Decoding (ID) - The instruction is decoded.
- Instruction Execution (EX) - The instruction is executed.

Flow through the pipeline

Figure A-1 shows how instructions move through the pipeline.

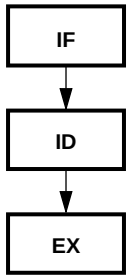


Figure A-1. Instruction Flow Through the Pipeline

The IF stage

In the IF stage, the CR16B fetches instructions into an internal register, the Instruction Register (IR), which holds one instruction at a time.

The ID stage

The IR feeds the instruction into the ID pipeline stage. At this point, a new instruction can be loaded into the IR. A new fetch request is issued whenever the bus is available and IR becomes available at the end of the next cycle.

The EX Stage

When execution of an instruction is completed, the decoded instruction (if there is one) is transferred to the Decoded Instruction Register (DIR), which holds one decoded instruction at a time.

The operations performed during execution depend on the instruction.

If it is an arithmetic or logic instruction:

- The Arithmetic/Logic Unit (ALU) or the shifter computes the result of the instruction.
- The result is written to the destination register.

If it is a load instruction:

- The ALU computes the effective memory address.
- The memory operand is read.
- The memory operand is written to the destination register.

If it is a store instruction:

- The ALU computes the effective memory address.
- The source operand is written to memory.

If it is a bit manipulation instruction:

- The ALU computes the effective memory address.
- The memory operand is read.
- The bit manipulation operation is done
- The result is written to memory.

If it is a branch or jump instruction:

- The ALU computes the target address.
- The target address is written to the PC register.

If it is a compare and branch instruction:

- The ALU performs the operand compare operation
- The ALU computes the target address.
- The target address is written to the PC register, according to the result of the compare.

Source operands are read and results are written, only in the EX stage. The instruction latency, when there are no delays, is usually three clock cycles.

The following sections describe most of the delays that may occur during execution of a program and which should be considered when evaluating a program's execution time.

A.3 EXECUTION DELAYS

Fetch delays It takes two bus cycles to fetch a double-word instruction from memory, because only one word can be fetched in each cycle. Decoding of the instruction, i.e., the ID stage, is delayed until both of its words have been fetched, i.e., are completely in the IF stage.

Therefore, if no instruction is in the ID stage when a double-word instruction fetch begins, a fetch delay occurs until the instruction currently being fetched can progress to the ID stage.

Branch delays Upon execution of a non-conditional branch or jump instruction, or a conditional branch or jump instruction that is true, subsequent instructions in the sequence may already have entered the pipeline, i.e., been fetched, and possibly decoded. These instructions are discarded, and clock cycles are added in which the target address for the branch instruction is fetched and decoded. These added clock cycles are called a branch delay.

For example, even if no wait states are needed while the memory from which instructions are fetched is accessed, the overall delay for a branch is two or three cycles (depending on the length of the target instruction).

Data delays A data delay occurs whenever the contents of a memory location are loaded into a register (using a load instruction) or when the contents of a register are stored in memory (using a store instruction). The length of the delay, i.e., the number of clock cycles that must be added, depends on the CPU, system configurations and the alignment of the data.

Load delays A load instruction is executed in two three or five cycles. In the first cycle the effective memory address is calculated and sent on the address bus. When memory is accessed without wait states, the data is returned on the data bus and stored in the appropriate register in the next cycle.

A load delay occurs in the following cases:

- When a byte to be accessed is not the least significant byte of an aligned word, i.e., the byte is not word aligned, the execution unit aligns it in an additional clock cycle. Execution takes three clock cycles.
- When a word to be accessed is not word-aligned, it is accessed in two consecutive bus cycles. One byte is accessed in each bus cycle and data is aligned after each bus cycle. Execution takes five clock cycles.
- When the main bus is already busy with an off-chip instruction fetch request, the load operation is deferred.

Store delays

A store instruction is executed in two or three cycles. In the first cycle, the effective memory address is calculated and sent on the address bus. In the next cycle, the contents of the register are sent on the data bus. All data is aligned as required, without performance penalties.

A store delay occurs in the following cases:

- When a word to be stored is not word-aligned, it is stored in two consecutive bus cycles. One byte is accessed in each bus cycle. Execution takes three clock cycles.
- When the main bus is already busy with an off-chip instruction fetch request, the store operation is deferred.

Serialized execution delays

When a serializing instruction is executed, all instructions residing in the IF stage and in the ID stage are discarded. Instructions that follow are not fetched from memory until the execution is complete. This causes a delay while the instructions following the serializing instruction are fetched.

See “Serializing Operations” on page 4-9 for more information on serializing instructions.

Long instructions execution delays

Some of the new CR16B instructions take more than 1 or 2 clocks to execute, since they perform a read-modify-write operation on memory, and execute complex functions on the data. These instructions, when encountered, may add delay to the total execution time, and to special cases, like exceptions and interrupt timings.

A.4 INSTRUCTION EXECUTION TIMING

The following sections specify the execution times in clock cycles, and other considerations that affect the total time required to execute each type of instruction. For more information about clock cycles that are added in the EX stage, see “Execution Delays” on page A-4

Arithmetic instructions

Table A-1. Execution Times for Arithmetic Instructions

Instruction	Clock Cycles in EX Stage (Zero Wait States)
ADDi, **ADDCi**, **ADDUi**, **ANDi**, **ASHUi**, **CMPi**, **LSHi**, **MOVi**, **MOVX**, **MOVZ**, **ORi**, **Scond**, **SUBCi**, **SUBi**, **NOP**, **TBIT**(*imm*/*reg-reg*) and **XORi**.	1
MOVD	2
MULB	4
MULW	8

Instruction	Clock Cycles in EX Stage (Zero Wait States)
MULSB	5
MULSW	15
MULUW	15

Load and store Instructions

Load or store instructions may be stalled in the EX stage for additional cycles while an instruction fetch is in process. This may occur when instructions are fetched from memory that requires wait states, e.g., off-chip memory.

In this case, additional clock cycles may be added to the number shown in the table. The number of clock cycles to add for this reason, depends on the speed at which memory can be accessed.

When zero wait states are used, the number of additional cycles that are needed to execute each load or store instruction depends on operand alignment on the word boundary.

Table A-2. Execution Times for Load and Store Instructions

Instruction	Clock Cycles in EX Stage (Zero Wait States)	Condition	Bus Accesses in EX Stage
LOADi	2	Minimum	1
	3	Byte access, not word aligned	1
		Word access, not word aligned	2
STORii	2	Minimum / Byte	1
		Word access, not word aligned	2

PUSH, POPrt, LOADM and STORM instructions

These instructions access the memory a multiple number of times, depending on the number of registers needing save or restore. Also, these instruction perform a pointer adjustment, which requires an extra clock in the store instructions. Each such access may be stalled in the EX stage for additional cycles while an instruction fetch is in process. This may occur when instructions are fetched from memory that requires wait states, e.g., off-chip memory.

In this case, additional clock cycles may be added to the number shown in the table. The number of clock cycles to add for this reason, depends on the speed at which memory can be accessed.

When zero wait states are used, the number of additional cycles that are needed to execute the instructions depend on the operation length itself, and on operand alignment on the word boundary.

Table A-3. Execution Times for PushM and PopM Instructions

Instruction	Clock Cycles in EX Stage (Zero Wait States)	Memory Address / Stack Alignment	Bus Accesses in EX Stage
PUSH[N]	N+1	Aligned	N
	3N	Not word aligned	2N
STORM[N]	N+2	Aligned	N
	3N+1	Not word aligned	2N
LOADM[N] / **POP[N]**	N+2	Aligned	N
	6N+1	Not word aligned	2N
+RET (for POP)	+1	Always	-

N = number of registers to load or store (1 - 4)

Memory bit manipulation instructions

Bit manipulation instructions perform a read-modify-write cycle on memory operands. As a result, the general timing for such an instruction is based on Load timing, execution timing, and some Store timing (somewhat shortened because of internal parallelism). Just as in load and store, the instructions may be stalled in the EX stage for additional cycles while an instruction fetch is in process. This may occur when instructions are fetched from memory that requires wait states, e.g., off-chip memory.

In this case, additional clock cycles may be added to the number shown in the table. The number of clock cycles to add for this reason, depends on the speed at which memory can be accessed.

When zero wait states are used, the number of additional cycles that are needed to execute the instructions depend on the operation length itself, and on operand alignment on the word boundary.

Table A-4. Execution Times for Bit Manipulation Instructions

Instruction	Clock Cycles in EX Stage (Zero Wait States)	Condition	Bus Accesses in EX Stage
SBITi/CBITi	4	Always	2
TBITi	3	Always	1

Control instructions

Some of the control instructions listed in Table A-5 also cause a pipeline flush and serialized execution of the next instruction. This delays execution, because clock cycles must be added to fetch and decode the instructions that follow the serializing instructions.

Table A-5. Execution Times for Control Instructions

Instruction(s)	Clock Cycles in EX Stage (Zero Wait States)	Pipeline Flush Condition
LPR	2	Always
SPR	2	Always
JUMP, **BAL**, **JAL**, **BR**	1	Always
Bcond, **Jcond**	1	When condition is true
Compare & Branch (**BCond0i**, **BCond1i**)	2	When condition is true
RETX	7	Always
WAIT, **EIWAIT**	1	Always, by interrupt
EI, **DI**	1	Never

Interrupts and traps

The interrupt or trap latency of the CR16B for a given interrupt or trap, in clock cycles, is the sum of the following:

- the longest execution time for a load or store instruction, or the specific long CR16B instruction interrupted (**SBIT**/**CBIT**, **TBITi**, **PUSH**, **POP**, **LOADM**, **STORM**)
- the time in clock cycles shown in Table A-6 for the specific exception
- the time it takes to fetch the first instruction in the interrupt handler
- additional clock cycles required as a result of wait states on the bus, hold requests, disabled interrupts or interrupt nesting

The execution time for the **MULi**, **MULUW** and **MULSB/W** instructions should not be used to calculate interrupt or trap latency. This is because, when an interrupt is detected while a **MULi** instruction, or one of the above, is in the EX pipeline stage, execution of **MULi**, **MULUW** and **MULSB/W**, does not continue to completion. Instead, execution of the **MULi** instruction or the others is suspended, and interrupt processing starts immediately. After interrupt processing is complete, execution of **MULi**, **MULUW** and **MULSB/W**, starts again, from the beginning.

Table A-6. Execution Times for Interrupts and Traps

Interrupt or Trap	Clock Cycles in EX Stage	
	Small Memory Model (16-bit Dispatch Table)	**Large Memory Model (32-bit Dispatch Table)**
INT	13	16
NMI and ISE	12	15
TRC	11	14
DBG	TBD	TBD
SVC, DVZ, FLG, BPT and UND	10	13

Appendix B
INSTRUCTION SET ENCODING

This appendix describes instruction encoding. Most instructions are encoded using one of the basic instruction formats. Where formats for instructions differ from the basic formats, e.g., load and store instructions, branch instructions and jump instructions, those differences are described separately.

Tables at the end of this Appendix summarize this instruction encoding information.

B.1 INTRODUCTION

Instructions may have zero, one or two operands, and are encoded using two or four bytes. All instructions must be word-aligned.

The basic structure of a two-operand instruction is shown below:

15 14	13	12 9	8 5	4 1	0
op code	i	op code	operand 2	operand 1	

Two, or three, bits code the operation (op code in bits 14, 15 and sometimes bit 0). One bit indicates the operation length (i in bit 13). Four bits (bits 9 through 12) may further specify the operation, or be used for a displacement value. Eight bits (bits 1 through 4 and bits 5 through 8) specify two instruction operands.

Bit 0 Bit 0 is used to extend other fields, e.g., op code or the first operand. See each format for more details.

Bits 1-4 When bits 1 through 4 (operand 1) specify a general-purpose register, it is usually the source register. This field may also contain a vector, an opcode extension, a constant (immediate) value, or a displacement value. If the constant or displacement value does not fit in the space allotted to it (its length is medium), it may be encoded in the next two bytes. See the following format descriptions for details.

Bits 5-8 When bits 5 through 8 (operand 2) specify a general-purpose register, it is usually the destination register. This field may also specify other special instruction options, such as a condition or a dedicated processor register, depending on the instruction.

Bits 9-12 Bits 9 through 12 may contain the operation code and/or a displacement value.

Bit 13 Bit 13 indicates the integer operation length (i). If i = 0, it is a byte (8-bit) operation; if i = 1, it is a word (16-bit) operation.

Bits 14-15 Bits 14 and 15, and sometimes bit 0, specify an operation code. Often, other bits are used with this op code to further specify the operation.

B.2 INSTRUCTION FORMATS

Most instructions use one of the basic formats described in the next section. In addition, load and store instructions, branch instructions (**BR**, **Bcond** and **BAL**) bit manipulation instructions, store-immediate, **MULSi**/**MULUW** instructions, **push**/**pop** instructions, and jump instructions (**JUMP**, **Jcond** and **JAL**) each use a different format. The following sections describe these formats.

B.2.1 Basic Instruction Formats

The **ADDi**, **ADDCi**, **ADDUi**, **ANDi**, **ASHUi**, **CMPi**, **LSHi**, **MOVi**, **MULi**, **ORi**, **SUBi**, **SUBCi**, **TBIT** and **XORi** instructions use one of the basic formats described in this section. The format used depends on the operands.

Register to register operations The format for instructions with two general-purpose register operands is shown below:

15 14	13	12 9	8 5	4 1	0
0 1	i	op code	dest reg	src reg	1

Short immediate to register operations A short immediate value is one that fits in the space provided in a 2-byte basic instruction format. The value −16 and all values in the range −14 through 15 can be encoded in this format.

The basic format for instructions that have short immediate values as operands is shown below. The core sign-extends the value in bits 0 through 4 (*imm*) to form a 16-bit immediate operand.

15 14	13	12 9	8 5	4 0
0 0	i	op code	dest reg	imm

Medium immediate to register operations

An immediate value that does not fit in the space allocated for the first operand in a 2-byte format, is a medium value. The signed 16-bit medium value is placed (encoded) in the next two bytes. All values in the range −32768 through 32767 can be encoded in this format.

The basic format, when the first operand is a medium immediate value (*imm*), is shown below:

31 16	15 14	13	12 9	8 5	4 0
imm	0 0	i	op code	dest reg	1 0 0 0 1

Compare and Branch special case

The Compare and Branch instruction, (**BEQ0/1i**, **BNE0/1i)** is a special case of the above, since the destination-registers are limited to a subset and also holds a portion of the opcode, and since the immediate field is a short one (5 bits) but with an implied value of 0 on bit 0. The registers available as base are R0, R1, R8, R9, and the immediate value is between 0 and 30 (2 x *imm*).

15	14	13	12 9	8	7 6	5	4 1	0
0	0	i	op code	rb3	op-ext	rb0	imm4-1	1

MOVD immediate special case

The **MOVD** instruction, when used with an immediate argument ($imm21) is also a special case of the above, since the immediate operand is of 21 bits length.

31 16	15 14	13	12 9	8 5	4	3 1	0
imm (I_{15} - I_0)	0 1	1	0 0 1 I_{20}	dest pair	I_{16}	(I_{19} - I_{17})	0

Instructions with special or no operands

The **DI**, **EI**, **EXCP**, **LPR**, **MOVXB**, **MOVZB**, **RETX**, **Scond**, **SPR**, **EIWAIT**, **MULSW**, **MULUW**, **MULSB**, **PUSH**, **POPrt**, **LOADM**, **STORM** and **WAIT** instructions either have unique operands or no operands at all. These instructions use the format shown below:.

15 14	13	12 9	8 5	4 1	0
0 1	1	op code	procreg/cond/reg /reg-pair/opcode-ext	reg/vector/imm/ opcode-ext	0

B.2.2 Load and Store Instructions

The **LOADi** and **STORi** instructions use the same formats. However, their op codes in bits 14 and 15 differ, and they specify different registers (reg in bits 5 through 8).

For the **LOADi** instruction, the op code (bits 14 and 15) is 10 and the reg field identifies the destination register. For the **STORi** instruction, the op code is 11 and the reg field identifies the source register.

Instruction	op code	reg
Load	1 0	Destination Register
Store	1 1	Source Register

The format to use for load and store instructions depends on the addressing mode and on the length of their displacement values. See also “Addressing Modes” on page 2-15.

Relative Addressing Mode

Short displacement values

A short displacement value fits within the field allotted for the displacement value in a 2-byte format (bit 0 and bits 9 through 12). This applies to any odd or even displacement value in the range 0 through 15, or any even displacement value in the range 16 through 30.

During execution, the core zero-extends the displacement field to 21 bits.

The format for load and store instructions, when the displacement value is short, is shown below:.

15 14	13	12 9	8 5	4 1	0
op code	i	disp (d_4–d_1)	reg	base reg	d_0

Medium displacement values

A medium displacement value is one that does not fit in a 2-byte format. In this case, the 18-bit displacement value is encoded by using two additional bytes. This format allows a displacement of either −128 Kbyte to 128K −1, or of 0 to 256 Kbyte, but limits the indexing to be between 0 to 64K−1 bytes (16-bit register). This instruction can be used to access only the 0 - 256 Kbyte range.

The format for load and store instructions, when the addressing mode is relative and the displacement value is medium, is shown below:

31 16	15 14	13	12 11	10 9	8 5	4 1	0
disp (d_{15}–d_0)	op code	i	1 0	d_{17} d_{16}	reg	base reg	1

Far- Relative Addressing Mode

The addressing mode of load and store instructions is called far-relative when the base address is specified by a pair of adjacent registers.

The base-pair field may encode any general-purpose register except SP, i.e., R0-RA that contains the 16 least significant bits of the base address. The five most significant bits of the base address are taken from the next consecutive register. In this case, the entire 18-bit displacement value is encoded by using two additional bytes, forming a 21-bit address space.

The 18-bit displacement is sign-extended to form a 21-bit number, with permitted values of −128K to 128K−1.

The instruction can index the whole 2 Mbyte range (base-pair concatenation).

This instruction can be used to access the whole 2 Mbyte range. The Load/Store Format, Far-Relative, is shown below:

31 16	15 14 13	12 11	10	9	8 5	4 1	0
disp (d_{15}–d_0)	op code	i	1 1	d_{17}	d_{16}	reg	base-pair	1

Absolute Addressing Mode

The format for load and store instructions, when the addressing mode is absolute, is shown below. This mode is limited to accessing only the first 256 Kbyte range.

31 16	15 14	13	12 11	10	9	8 5	4 0
abs (d_{15}–d_0)	op code	i	1 1	d_{17}	d_{16}	reg	1 1 1 1 1

B.2.3 Branch Instructions

Branch instructions, i.e., **Bcond**, **BAL** and **BR**, specify the target address as a displacement from the address currently in the Program Counter (PC). The displacement value is interpreted as a signed integer. The target address is the value in the displacement field plus the address currently in the PC. These instructions exist in two formats - the small memory model and the large memory model. The small-memory-model **Bcond** and **BR** instructions with a medium displacement, and the small-memory-model **BAL** truncate address calculation at the 128K boundary, limiting execution to the first 128K of memory. The **Bcond** and **BR** instructions with a short displacement, the large-memory-model medium displacement **Bcond** and **BR**, and the large-memory-model **BAL** perform 21-bit address calculation, allowing execution in the whole 2M address range.

Since all instructions are word-aligned, in the **Bcond**, **BAL** and **BR** instructions, displacement values must be even.

BR and Bcond Instructions

The **BR** and **Bcond** instructions can be encoded in 2 or 4 bytes, depending on whether the displacement value is short or medium respectively. The core sign-extends short and medium branch displacement values to 21 bits, and after address calculation, it truncates the resulting address to either 17 bits (for medium displacement values in the small-memory-format), or to 21 bits (for short displacement values, or medium displacement values in the large-memory-format).

In the **Bcond** instruction, bits 5 through 8 specify the condition.

Short displacement values

Even displacement values in the range −256 through 254 are called short. Short values fit in nine bits provided in a 2-byte format. The displacement value is encoded in bits 0 through 4 (d_0 through d_4, respectively) and bits 9 through 12 (d_5 through d_8, respectively). The displacement is sign extended to 21 bits, and the whole address calculation is done at 21 bits, with the result truncated to 21 bits. This allows short branches throughout the 2M address space, with a wrap-around at the 2 Mbyte boundary.

The format of **BR** or **Bcond** instructions, with short displacement values, is shown below. Note that in this format, bit 0 (d_0) must be 0.

15 14	13	12 9	8 5	4 0
0 1	0	disp (d_8–d_5)	cond	disp (d_4–d_0)

Medium displacement values - small memory format

When the displacement value does not fit in the 2-byte format, it is called medium and encoded into two additional bytes as shown below. The displacement is sign extended to 21 bits, and the whole address calculation is done at 21 bits, with the result truncated to 17 bits. This forces conformance with the CR16A 'branch with medium displacement', which is limited to the first 128 Kbyte of memory, and wraps around at 128K. This conforms to the small-memory-model of the CR16B. Note that bit 16 (d_0) must be 0.

31 16	15 14	13	12 9	8 5	4	3 0
disp (d_{15}–d_0)	0 0	0	1 0 1 0	cond	d_{16}	1 1 1 0

Medium displacement values - large memory format

The large-memory-format of the **BR** and **Bcond** instructions expand the CR16B branch capability to the full 2 Mbyte (21 address bits) address space. They are always encoded in 4 bytes, allowing a full 21-bit displacement value (large-memory-model). Address calculations are truncated to 21 bits.

As in the small memory format, bits 5 through 8 specify the condition.

The format of **BR** or **Bcond** instructions, with large-memory-model medium displacement values, is shown below. Note that in this format, bit 0 (d_0) is assumed 0.

31 17	16	15	14	13	12 9	8 5	4	3 1	0
disp (d_{15}–d_1)	d_{20}	0	1	1	1 0 1 0	cond	d_{16}	d_{19}–d_{17}	0

The BAL Instruction

The BAL instruction - small memory format

For the **BAL** instruction in the small-memory-format, the displacement value is encoded as shown below. The displacement is sign extended to 21 bits, and the whole address calculation is done at 21 bits, with the result truncated to 17 bits. This forces conformance with the CR16A '**bal**' instruction, which is limited to the first 128 Kbyte of memory, and wraps around at 128K (small CR16B-memory-model). Bits 5 through 8 specify the link register. As in the **BR** and **Bcond** instructions, bit 16 (d_0) must be 0.

31 16	15 14	13	12 9	8 5	4	3 0
disp (d_{15}–d_0)	0 0	1	1 0 1 0	link reg	d_{16}	1 1 1 0

The BAL instruction - large memory format

For the large memory format of the **BAL** instruction, the displacement value is encoded as shown below. This command allows branch and link operation throughout the whole 2M address space, enhancing the CR16A **BAL** instruction.

The return address is saved in a pair of adjacent registers. Bits 5 through 8 specify the link-pair selection. The link-pair field may encode any general-purpose register except SP and RA, i.e., R0-ERA that contains bits 1 through 16 of the address. The four most significant bits are stored in the next consecutive register, right justified.

As in the **BR** and **Bcond** instructions, bit d0 is assumed 0.

31 17	16	15 14	13	12 9	8 5	4	3 1	0
disp (d_{15}–d_1)	d_{20}	0 1	1	1 0 1 1	lnk-pair	d_{16}	d_{19}–d_{17}	0

B.2.4 Jump Instructions

The JUMP and Jcond instructions - small memory format

The small-memory-format of the **JUMP** and **Jcond** instructions use the format shown below. This instruction is limited to the lower 128 Kbyte addresses (small-memory-model).

15 14	13	12 9	8 5	4 1	0
0 1	0	1 0 1 0	cond	target reg	1

The JMP and Jcond instructions-large memory format

The large memory format of the **JMP** and **Jcond** instructions use the format shown below. This format uses a pair of adjacent registers to hold the jump address (21 bits, bit a0 assumed to be always 0). The target-pair field may encode any general-purpose register except SP and RA, i.e., R0-ERA that contains bits 1 through 16 of the address. The four most significant bits are stored in the next consecutive register, right justified.

15 14	13	12 9	8 5	4 1	0
0 0	0	1 0 1 1	cond	target-pair	1

The JAL instruction-small memory format

In small-memory-format of the **JAL** instruction, bits 5 through 8 specify the link register. It is encoded as shown below. This command is limited to the lower 128 Kbyte addresses (small-memory-model)

15	14	13	12			9	8	5	4	1	0
0	1	1	1	0	1	0	link reg		target reg		1

The JAL instruction - large memory format

In the large memory format of the **JAL** instruction, as in **BAL** and **JUMP**, a pair of adjacent registers is used as a link address, and another pair is used as a jump address. Bits 5 through 8 specify the link-pair, and bits 4 through 1 specify the target-pair. Registers R0-ERA can be selected as link-pairs or target-pairs. This command covers the whole 2 Mbyte address space (large-memory-model). It is encoded as shown in below:

15	14	13	12			9	8	5	4	1	0
0	0	0	1	0	1	1	link-pair		target-pair		0

B.2.5 Bit Manipulation and Store of Immediate-Value Instructions

The bit manipulation instructions (**SBIT**, **CBIT**, **TBIT**) and the store immediate version of the **STOR** instruction (**STORi $imm4,<>)** use part of the register field to expand the opcode set, thus allowing to encode more commands, at the expense of a narrower range of base registers. The instructions allow only a set of four registers to be used as base (R0, R1, R8, R9), with three addressing formats to access memory. The first is a register relative mode, with no displacement, which is encoded in a two-byte command. The second is a register-relative with a 16-bit displacement, which is encoded in a four-byte command. The third is a 18-bit absolute address mode, which is also encoded in a four-byte command.

Register-relative with no displacement memory format

The format is shown below:

15	14	13	12			9	8		5	4	1	0
0	1	i	0	0	1	0	bs1	ex-op	bs0	bit-num/Imm		1

This mode is limited to accessing only the first 64 Kbyte of memory. The parameters for these instructions are:

1. bs1,bs0
   Base register selection bits; bs1,bs0 = [R0,R1,R8,R9]

2. ex-op
   Opcode expansion field; these bits are used to differentiate between **SETB**, **CLRB**, **STRM** and **TSTB**.

3. bit-num/Imm
   This field is either used to select which bit is to be the target of the bit-operation, or as a 4-bit immediate data for the **STORi $imm4,<>**.

Register-relative with 16-bit displacement memory format

The format is shown below:

31 16	15	14	13	12	11	10	9	8		5	4 1	0
disp (d_{15}–d_0)	0	0	i	0	0	1	0	bs1	ex-op	bs0	bit-num/Imm	1

The displacement in this case is unsigned, and limited to the range of 0 to 64K − 1 bytes. This mode has only limited access to the memory range over 64 Kbytes

18-bit absolute memory format

The format is shown below:

31 16	15	14	13	12	11	10	9	8		5	4 1	0
ABS (ad_{15}–ad_0)	0	0	i	0	0	1	0	ad_{17}	ex-op	ad_{16}	bit-num/Imm	0

The displacement in this case is absolute; this mode can access the first 256 Kbyte of memory

B.3 UNDEFINED OP CODES

The following op codes cause an undefined instruction trap.

Table B-1. Undefined Op Codes

Op Code	i	Op Code	Operand 2	Operand 1	Op Code or Operand 1	Comment
0 1	1	0 1 1 0	X X X X	0 X X X	0	reserved
0 1	1	1 1 1 1	0000-1110	0 0 1 1	0	reserved
0 1	1	1 1 1 1	X X X X	0 0 1 0	0	reserved
0 1	1	1 1 1 1	1 X X X	0 0 1 0	0	reserved
0 1	1	1 1 1 1	X X X X	1 0 1 X	0	reserved
0 1	1	1 1 1 1	X X X X	X 0 0 X	0	reserved
0 1	1	1 1 1 0	X X 0 X	X X X X	0	reserved

B.4 CR16B INSTRUCTION SET SUMMARY

Table B-2. Notation Conventions for Instruction Set Summary

i	=	Operation length field 0 – Byte (8 bits) 1 – Word (16 bits)
abs	=	Absolute address
imm	=	Immediate value
immxx	=	xx bit Immediate value respectively
disp	=	Displacement value
dispxx	=	xx bit Displacement value respectively
dest	=	Destination
src	=	Source
SMM, LMM	=	Small Memory Model and Large Memory Model respectively

Rsrc, Rdest, Rlink, RLpair, Rbase, Rpair, Rtarget, Roffset = Source, destination, link, base, base pair, target or offset register, respectively.

0000 – R0
0001 – R1
0010 – R2
0011 – R3
0100 – R4
0101 – R5
0110 – R6
0111 – R7
1000 – R8
1001 – R9
1010 – R10
1011 – R11
1100 – R12
1101 – R13 , ERA
1110 – RA
1111 – SP

Rproc =

Dedicated CPU register
0001 – PSR
0101 – CFG
0011 – INTBASEL
0100 – INTBASEH
1011 – ISP
0111 – DSR
1001 – DCR
1101 – CARL
1110 - CARH

Condition code field

cond =	0000 – EQ	Equal	Z = 1
	0001 – NE	Not Equal	Z = 0
	1101 – GE	Greater than or Equal	N = 1 or Z = 1
	0010 – CS	Carry Set	C = 1
	0011 – CC	Carry Clear	C = 0
	0100 – HI	Higher than	L = 1
	0101 – LS	Lower than or the Same as	L = 0
	1010 – LO	Lower than	L = 0 and Z = 0
	1011 – HS	Higher than or the Same as	L = 1 or Z = 1
	0110 – GT	Greater Than	N = 1
	0111 – LE	Less than or Equal	N = 0
	1000 – FS	Flag Set	F = 1
	1001 – FC	Flag Clear	F = 0
	1100 – LT	Less Than	N = 0 and Z = 0

Exception vector (used by EXCP instruction)

vector =
0101 – SVC
0110 – DVZ
0111 – FLG
1000 – BPT
1010 – UND
1110 – DBG
others– reserved

PUSH/POP/LOADM/STORM register save/restore/load/store count

Rcnt =
00 – 1 register
01 – 2 registers
10 – 3 registers
11 – 4 registers

Table B-3. CompactRISC CR16B Addressing Calculations

Instructions	Addressing Options	Displacement Type	Target Address Calculation
BEQ0/1i, BNE0/1/i	dispu5	5-bit unsigned, even (0-30)	ITruncate_21[a] (PC + zext21[b](dispu5));
BR/BCond	disp9	9-bit signed	ITruncate_21[a] (PC + sext21[c](disp9))
BR/BCond, **BAL** (small memory model)	disp17	17-bit signed/unsigned. (-64k->64K-1 / 0->128K-1)	ITruncate_17[d] (PC + sext18[e](disp17))
BR/BCond, **BAL** (large memory model)	disp21	21-bit signed/unsigned. (-1M->1M-1 / 0->2M-1)	ITruncate_21[a] (PC + disp21)
JUMP, **JAL** (small memory model)	Rtarget	-----	ITruncate_17[d] (Rtarget << 1)
JUMP, **JAL** (large memory model)	(Rtarget+1, Rtarget)	-----	ITruncate_21[a] ((Rtarget+1,Rtarget) << 1)
LOADi, **STORi**	dispu5	5 bit unsigned (0-15; even 16-30);	DTruncate_18[f] (zext21[b](Rbase) + zext21[b](dispu5))
TBITi, **CBITi**, **SBITi**, **STORi $imm4**,**<>**	0(Rbase), disp16(Rbase)	16 bit unsigned	DTruncate_18[f] (zext21[b](Rbase) + z zext21[b](disp16))
TBITi, **CBITi**, **SBITi**, **STORi $imm4**,**<>**, **LOADi**, **STORi**	abs18	18 bit absolute	DTruncate_18[f](abs18)
LOADi, **STORi**	disp18(Rbase)	18 bit signed/unsigned. (-128k->128K-1 / 0-256K-1)	DTruncate_18[f] (zext21[b](Rbase) + sext21[c](disp18))
LOADi, **STORi** (far)	disp18 (Rbase+1,Rbase)	18 bit signed (-128k->128K-1)	DTruncate_21[g] (Rpair + sext21[c](disp18))

a. ITruncate_21: Perform calculation on 21 bit ALU. (Wrap at 2 MByte). Address bit 0 is cleared to 0 on all instruction addressing calculations.
b. zext21: zero extend the argument to 21 bits.
c. sext21: sign extend the argument to 21 bits.
d. ITruncate_17: Truncate the address calculation result at 17 bits - 128 Kbyte wrap around. Address bits 17 through 20 and bit 0 are cleared to 0.
e. sext18: sign extend the argument to 18 bits.
f. DTruncate_18: Truncate the address calculation result at 18 bits - 256 Kbyte wrap around. Address bits 18 through 20 are cleared to 0.
g. DTruncate_21: Truncate the address calculation result at 21 bits - 2 Mbyte wrap around (full 21 bit ALU).

Table B-4. Instruction Encoding

Mnemonic	Operands	15	14	13	12			9	8		5	4			1	0
MOVES																
MOVi	Rsrc, Rdest	0	1	i	1	1	0	0	Rdest			Rsrc				1
	imm, Rdest	0	0	i	1	1	0	0	Rdest			imm				i_0
MOVXB	Rsrc, Rdest	0	1	1	0	1	0	0	Rdest			Rsrc				0
MOVZB	Rsrc, Rdest	0	1	1	0	1	0	1	Rdest			Rsrc				0
MOVD	imm21, (Rdest+1,Rdest)	0	1	1	0	0	1	I_{20}	Rdest			I_{16}	I_{19}	I_{18}	I_{17}	0
INTEGER ARITHMETIC																
ADDi	Rsrc, Rdest	0	1	i	0	0	0	0	Rdest			Rsrc				1
	imm, Rdest	0	0	i	0	0	0	0	Rdest			imm				i_0
ADDUi	Rsrc, Rdest	0	1	i	0	0	0	1	Rdest			Rsrc				1
	imm, Rdest	0	0	i	0	0	0	1	Rdest			imm				i_0
ADDCi	Rsrc, Rdest	0	1	i	1	0	0	1	Rdest			Rsrc				1
	imm, Rdest	0	0	i	1	0	0	1	Rdest			imm				i_0
MULi	Rsrc, Rdest	0	1	i	0	0	1	1	Rdest			Rsrc				1
	imm, Rdest	0	0	i	0	0	1	1	Rdest			imm				i_0
MULSB	Rsrc, Rdest (signed)	0	1	1	0	0	0	0	Rdest			Rsrc				0
MULSW	Rsrc, (Rdest+1,Rdest) (signed)	0	1	1	0	0	0	1	Rdest			Rsrc				0
MULUW	Rsrc, (Rdest+1,Rdest) (unsigned)	0	1	1	1	1	1	1	Rdest			R_{s3}	0	0	R_{s0}	0
SUBi	Rsrc, Rdest	0	1	i	1	1	1	1	Rdest			Rsrc				1
	imm, Rdest	0	0	i	1	1	1	1	Rdest			imm				i_0
SUBCi	Rsrc, Rdest	0	1	i	1	1	0	1	Rdest			Rsrc				1
	imm, Rdest	0	0	i	1	1	0	1	Rdest			imm				i_0
INTEGER COMPARISON																
CMPi	Rsrc, Rdest	0	1	i	0	1	1	1	Rdest			Rsrc				1
	imm, Rdest	0	0	i	0	1	1	1	Rdest			imm				i_0
BEQ0i	Rdest, disp5	0	0	i	1	0	1	0	R_{d3}	0	0	R_{d0}	d4	d3	d2	d1	1
BEQ1i	Rdest, disp5	0	0	i	1	0	1	0	R_{d3}	0	1	R_{d0}	d4	d3	d2	d1	1
BNE0i	Rdest, disp5	0	0	i	1	0	1	0	R_{d3}	1	0	R_{d0}	d4	d3	d2	d1	1
BNE1i	Rdest, disp5	0	0	i	1	0	1	0	R_{d3}	1	1	R_{d0}	d4	d3	d2	d1	1
LOGICAL AND BOOLEAN																
ANDi	Rsrc, Rdest	0	1	i	1	0	0	0	Rdest			Rsrc				1
	imm, Rdest	0	0	i	1	0	0	0	Rdest			imm				i_0
ORi	Rsrc, Rdest	0	1	i	1	1	1	0	Rdest			Rsrc				1
	imm, Rdest	0	0	i	1	1	1	0	Rdest			imm				i_0
Scond	Rdest	0	1	1	0	1	1	1	cond			Rdest				0
XORi	Rsrc, Rdest	0	1	i	0	1	1	0	Rdest			Rsrc				1
	imm, Rdest	0	0	i	0	1	1	0	Rdest			imm				i_0

Table B-4. Instruction Encoding (Continued)

Mnemonic	Operands	15	14	13	12			9	8			5	4			1	0
SHIFTS																	
ASHUi	Rsrc, Rdest	0	1	i	0	1	0	0	Rdest				Rsrc				1
	imm, Rdest	0	0	i	0	1	0	0	Rdest				imm				i_0
LSHi	Rsrc, Rdest	0	1	i	0	1	0	1	Rdest				Rsrc				1
	imm, Rdest	0	0	i	0	1	0	1	Rdest				imm				i_0
BITS																	
TBIT	Roffset, Rsrc	0	1	1	1	0	1	1	Rsrc				Roffset				1
	imm, Rsrc	0	0	1	1	0	1	1	Rsrc				imm				i_0
TBITi	imm, 0(Rbase = 0,1,8,9)	0	1	i	0	0	1	0	R_{b3}	1	0	R_{b0}	imm (position)				1
	imm, disp16(Rbase = 0,1,8,9)	0	0	i	0	0	1	0	R_{b3}	1	0	R_{b0}	imm (position)				1
	imm, abs18	0	0	i	0	0	1	0	d_{17}	1	0	d_{16}	imm (position)				0
CBITi	imm, 0(Rbase = 0,1,8,9)	0	1	i	0	0	1	0	R_{b3}	0	0	R_{b0}	imm (position)				1
	imm, disp16(Rbase = 0,1,8,9)	0	0	i	0	0	1	0	R_{b3}	0	0	R_{b0}	imm (position)				1
	imm, abs18	0	0	i	0	0	1	0	d_{17}	0	0	d_{16}	imm (position)				0
SBITi	imm, 0(Rbase = 0,1,8,9)	0	1	i	0	0	1	0	R_{b3}	0	1	R_{b0}	imm (position)				1
	imm, disp16(Rbase = 0,1,8,9)	0	0	i	0	0	1	0	R_{b3}	0	1	R_{b0}	imm (position)				1
	imm, abs18	0	0	i	0	0	1	0	d_{17}	0	1	d_{16}	imm (position)				0
PROCESSOR REGISTER MANIPULATION																	
LPR	Rsrc, Rproc	0	1	1	1	0	0	0	Rproc				Rsrc				0
SPR	Rproc, Rdest	0	1	1	1	0	0	1	Rproc				Rdest				0
JUMPS AND LINKAGE																	
Bcond	disp9	0	1	0	d_8	d_7	d_6	d_5	cond				d_4	d_3	d_2	d_1	0
	disp17 - SMM	0	0	0	1	0	1	0	cond				d_{16}	1	1	1	0
	disp21 - LMM	0	1	1	1	0	1	0	cond				d_{16}	d_{19}	d_{18}	d_{17}	0
BAL	Rlink, disp17 - SMM	0	0	1	1	0	1	0	Rlink				d_{16}	1	1	1	0
	(Rlink+1,Rlink), disp21 - LMM	0	1	1	1	0	1	1	Rlink				d_{16}	d_{19}	d_{18}	d_{17}	0
BR	disp9	0	1	0	d_8	d_7	d_6	d_5	1	1	1	0	d_4	d_3	d_2	d_1	0
	disp17 - SMM	0	0	0	1	0	1	0	1	1	1	0	d_{16}	1	1	1	0
	disp21 - LMM	0	1	1	1	0	1	0	1	1	1	0	d_{16}	d_{19}	d_{18}	d_{17}	0
EXCP	vector	0	1	1	1	1	0	1	1	1	1	1	vector				0
Jcond	Rtarget - SMM	0	1	0	1	0	1	0	cond				Rtarget				1
	(Rtarget+1,Rtarget) - LMM	0	0	0	1	0	1	1	cond				Rtarget				1
JAL	Rlink, Rtarget - SMM	0	1	1	1	0	1	0	Rlink				Rtarget				1
	(Rlink+1,Rlink), (Rtarget+1,Rtarget) - LMM	0	0	0	1	0	1	1	Rlink				Rtarget				0
JUMP	Rtarget - SMM	0	1	0	1	0	1	0	1	1	1	0	Rtarget				1
	(Rtarget+1,Rtarget) - LMM	0	0	0	1	0	1	1	1	1	1	0	Rtarget				1
RETX		0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0
PUSH	imm, Rsrc	0	1	1	0	1	1	0	0	0	Rcnt[a]		Rsrc				0
POP	imm, Rdest	0	1	1	0	1	1	0	0	1	Rcnt[a]		Rsrc				0
POPRET	imm, Rdest - SMM	0	1	1	0	1	1	0	1	0	Rcnt[a]		Rsrc				0
POPRET	imm, Rdest - LMM	0	1	1	0	1	1	0	1	1	Rcnt[a]		Rsrc				0

Table B-4. Instruction Encoding (Continued)

Mnemonic	Operands	15	14	13	12			9	8			5	4			1	0
LOAD AND STORE																	
LOADi	disp(Rbase), Rdest	1	0	i	d_4	d_3	d_2	d_1	Rdest				Rbase				d_0
	disp(Rbase), Rdest	1	0	i	1	0	d_{17}	d_{16}	Rdest				Rbase				1
	disp(Rpair+1, Rpair), Rdest	1	0	i	1	1	d_{17}	d_{16}	Rdest				Rpair				1
	abs, Rdest	1	0	i	1	1	d_{17}	d_{16}	Rdest				1	1	1	1	1
LOADM	imm	0	1	1	1	1	1	1	0	0	Wcnt[a]		0	0	1	0	0
STORi	Rsrc, disp(Rbase)	1	1	i	d_4	d_3	d_2	d_1	Rsrc				Rbase				d_0
	Rsrc, disp(Rbase)	1	1	i	1	0	d_{17}	d_{16}	Rsrc				Rbase				1
	Rsrc, disp(Rpair+1, Rpair)	1	1	i	1	1	d_{17}	d_{16}	Rsrc				Rpair				1
	Rsrc, abs	1	1	i	1	1	d_{17}	d_{16}	Rsrc				1	1	1	1	1
	imm4, 0(Rbase = 0,1,8,9)	0	1	i	0	0	1	0	R_{b3}	1	1	R_{b0}	imm4 (data)				1
	imm4, disp16(Rbase= 0,1,8,9)	0	0	i	0	0	1	0	R_{b3}	1	1	R_{b0}	imm4 (data)				1
	imm4, abs18	0	0	i	0	0	1	0	d_{17}	1	1	d_{16}	imm4 (data)				0
STORM	imm	0	1	1	1	1	1	1	0	1	Wcnt[a]		0	0	1	0	0
MISCELLANEOUS																	
DI		0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	0
EI		0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0
NOP		0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
WAIT		0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
EIWAIT		0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0

a. Wcnt = imm -1. The encoded count value = number_of_words (or registers) - 1.

Appendix C
STANDARD CALLING CONVENTIONS

C.1 CALLING CONVENTION

The primary goal of standard routine-calling conventions is to enable the routines of one module to communicate with routines in other modules, even if they are written in different programming languages.

The calling convention is defined as part of the CompactRISC architecture, and is enforced and supported by the CompactRISC Development Tools. The calling convention consists of a set of rules which form a handshake between different pieces of code (subroutines), and define how control is transferred from one to another. It thus defines a general mechanism for calling subroutines and returning from subroutines.

In summary, calling a routine consists of the following steps:

1. Parameter values are computed, and placed in registers or on the stack.
2. The **BAL** or **JAL** instruction is executed.
3. The called function allocates its area on the run-time stack, and saves the values of the *safe* registers it plans to use.
4. The called function executes, i.e., it computes the return value, and stores it in a register.
5. The called function restores the safe registers, and de-allocates its stack section.
6. Control is returned to the calling functions by means of a **JUMP** instruction.

The rest of this appendix describes this procedure in greater detail.

C.1.1 Calling a Subroutine

The **BAL** or **JAL** instruction is used to call a subroutine. Each of these instructions performs two operations:

- It saves the address of the following instruction (the value of the Program Counter register) in a specified general-purpose register pair (for large model), or in a single general-purpose register (for small model).

The calling convention requires the use of (ERA, RA) register pair (for large model), or the RA register (for small model). This register pair (or single register) is used to store the return address.

- Transfers control to a specified location in the program (the subroutine address).

Example 1 For large model:

```
bal   (era,ra), get_next  # call the subroutine "get_next"
```

or

```
jal   (era,ra), (r8,r7)   # call the subroutine whose address
                            is stored in the pair r7, r8
```

Example 2 For small model:

```
bal   ra, get_next        # call the subroutine "get_next"
```

or

```
jal   ra, r7              # call the subroutine whose
                            address is stored in r7
```

C.1.2 Returning from a Subroutine

The jump instruction is used to return from a subroutine. The Program jumps to the return address, which is stored in (ERA, RA) register pair (for large model), or the RA register (for small model), as follows:

For large model:

```
jump (era,ra)       # return to caller
```

For small model:

```
jump ra             # return to caller
```

C.2 CALLING CONVENTION ELEMENTS

This section describes conventions for how parameters are passed to a called subroutine, how values are returned, the program stack, and calling different types of general purpose registers.

C.2.1 Passing Parameters to a Subroutine

Qualifying arguments may be passed to the called subroutine by loading them into registers according to a predefined convention. The registers used are the integer registers R2, R3, R4, R5.

Qualifying Arguments

A qualifying argument may be one of the following types:

- Integer type, pointer type
- Aligned structure whose size is less than, or equal to, four bytes.

  32-bit long integer types and pointers are considered two-word structures, in this context. The least significant word of a multi-word structure is always stored first.

The Algorithm

The following algorithm determines how parameters are passed to a given routine:

1. The parameter list is scanned from left to right.
2. A qualifying argument is allocated to the next free register in ascending order, i.e., R2 is allocated before R3, etc. Multi-word structures use a register pair. The least significant word is allocated to the first register, and the most significant word to the next consecutive register.
3. If a parameter cannot be passed in a register (either because it is not qualified, or because the registers have been entirely allocated to previous parameters) then this parameter is passed on the stack, least significant byte first.

C.2.2 Returning a Value

A subroutine can return one value to its caller. The calling convention uses the R0 register for passing a short return value and the register pair R0 and R1 for passing a long (4-byte) return value, with the least significant word in R0.

For example, consider the following C code:

```
return 5;
```

The assembly code generated from this line is:

For large model:

```
movw $5, r0          # pass return value
jump (era,ra)        # return to caller
```

For small model:

```
movw $5, r0          # pass return value
jump ra              # return to caller
```

The only exception to this rule is a function that returns a structure. In this case, the calling function must store the address of a structure in R0. The called function then uses R0 as a pointer to store the resulting structure.

C.2.3 Program Stack

The program stack is a contiguous memory space that may be used by your program for:

- Allocating memory for local variables which are not in registers.
- Passing arguments in special cases. (See “Passing Parameters to a Subroutine” on page C-3.)
- Saving registers before calling a subroutine, or after being called. (See “Scratch Registers” on page C-5.)

The stack is a dynamic memory space which begins at a fixed location (stack bottom) and grows towards lower memory addresses. Its lowest address (also called top of stack) is changed dynamically and is pointed to by the Stack Pointer (SP) register.

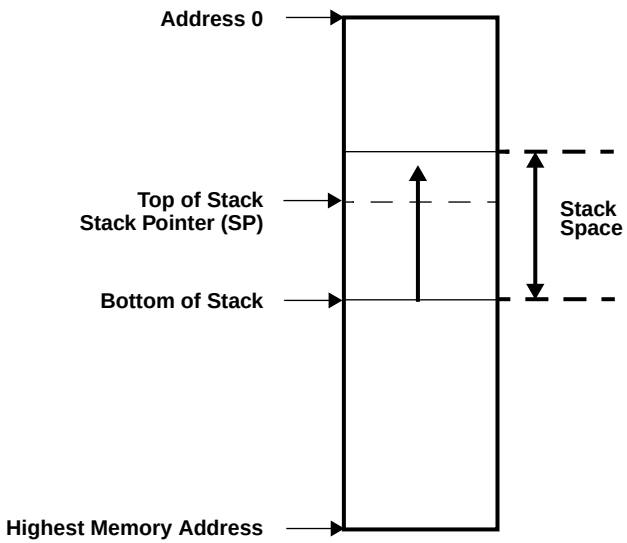


Figure C-1. The Program Stack

A subroutine can allocate space on the stack by decrementing the value of the SP register to adjust the top of stack. When this subroutine returns it must restore the SP to its previous value, thereby releasing the temporary space that it had occupied on the stack during its life-time.

The program stack always resides in the lowest 64 Kbytes of the CR16B address space.

C.2.4 Scratch and Safe Registers

According to the convention, CompactRISC general purpose registers may be used as scratch registers or safe registers.

Scratch Registers

Any of these registers can be freely modified by any subroutine without first saving a backup of their previous value. The caller cannot assume that their value will remain the same after a subroutine has returned. If for any reason the caller needs to keep this value, it is its responsibility to save the scratch register on the stack before calling the subroutine, and to restore it after the subroutine has returned.

Safe Registers

Before using any of these registers, a subroutine must first store its previous value on the stack. Upon returning to the caller the subroutine must restore this value. The caller can always assume that these registers will not be clobbered by any subroutine that it has called.

This calling convention defines R0 through R6 as scratch registers. All the other general purpose registers, including ERA, RA and SP, are safe.

Exception The interrupt/trap subroutine is an exception to the rule for using scratch registers. This kind of subroutine must always save and restore every scratch register that may be used during the interrupt trap. This is because there is no real caller. The interrupt, or trap, suspends another subroutine which is not aware of, or prepared for, this interception. To protect it, its scratch registers must be saved and restored so that the interrupt, or trap, is transparent.

Appendix D
COMPARISON OF CR16A AND CR16B

The CR16B architecture is an enhancement of the CR16A architecture. CR16B has two programming models: large and small. The small programming model is backward compatible with the CR16A, in the sense that CR16A programs can be compiled and executed under the CR16B small model, with no changes.

The purpose of this appendix is to help CR16A developers migrate to CR16B. It focuses on both the implications for applications using the large programming model, and on the differences between the small programming model of the CR16B and the CR16A programming model (mainly in the instruction set).

After you have familiarized yourself with the differences between the CR16A and CR16B instruction sets, you will be able to improve the performance and the code optimization of your existing applications.

D.1 REGISTER SETS

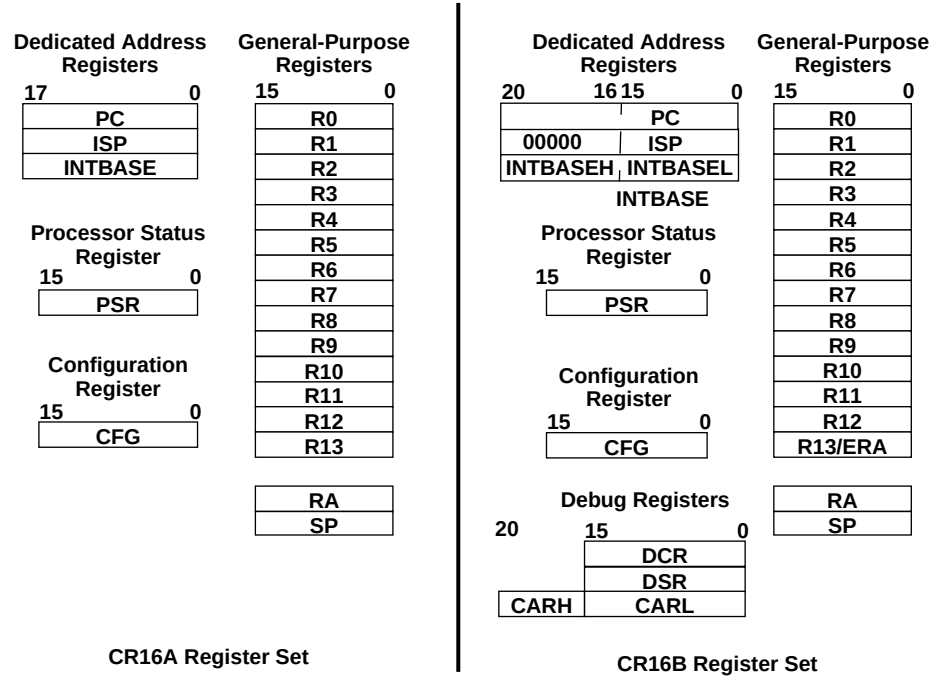


Figure D-1. The CR16A and CR16B Register Sets

D.1.1 General-Purpose Registers

The general-purpose registers are 16 bits wide in both the CR16A and the CR16B. Some of the new CR16B instructions can operate on specific general-purpose registers, whereas in the CR16A all general-purpose registers are equal for all instructions.

CR16A and the CR16B Small Programming Model

Both the CR16A and the CR16B small programming model make similar use of the general-purpose registers:

- Registers **r0** - **r13** are used for general purposes, such as holding variables, addresses or index values.
- The **SP** register is usually used as the program stack pointer.
- The **RA** register is usually used for storing the return address from a subroutine.

CR16B Large Programming Model

The CR16B large programming model makes the following use of the general-purpose registers:

- Registers **r0** - **r12** are used for general purposes, such as holding variables, addresses or index values.
- The **SP** register is usually used as the program stack pointer.
- The **ERA** and **RA** register pair is usually used for storing the return addresses from a subroutine.

Use of Specific General-purpose Registers by CR16B Instructions

In the CR16B only, several instructions operate on specific registers.

The **MULUW, BEQ1, BENEQ1, BEQ0, BNEQ0, CBIT, SBIT, TBIT** instructions and the **STORi** instruction (when its sources is an immediate operand) use a sub-set of the general-purpose registers set (**r0**, **r1**, **r8** and **r9**).

The **PUSH** and **POP** instruction make specific use of the SP register. The **POPRET** instruction makes specific use of **RA** (small model) or **ERA-RA**, in addition to the **SP** register.

The **LOADM** instruction makes specific use of registers **r0** and **r2-r5**.

The **STORM** instruction makes specific use of registers **r1** and **r2-r5**.

D.1.2 Dedicated Address Registers

The dedicated address registers, used by the CR16B to implement specific address functions, are 18 bits wide in the CR16A, and 21 bits wide in both models of the CR16B. See “Dedicated Address Registers” on page 2-7 for a detailed description of these registers.

CR16A Programming Model

The three dedicated address registers **PC**, **ISP** and **INTBASE**, are 18 bits wide. However, the most significant bit of the **PC**, and the two most significant bits of the **ISP** and **INTBASE** registers, are always cleared. Therefore, program code is limited to the lowest 128K of the address space and the interrupt stack and the interrupt dispatch table must reside in the lowest 64 Kbytes of the address space.

CR16B Small Programming Model

The three dedicated address registers **PC**, **ISP** and **INTBASE**, are 21 bits wide. However, the four most significant bits of the **PC**, and the five most significant bits of the **ISP** and **INTBASE** registers, are always cleared. Therefore, program code is limited to the lowest 128K of the address space and the interrupt stack and the interrupt dispatch table must reside in the lowest 64 Kbytes of the address space.

CR16B Large Programming Model

The three dedicated address registers, **PC**, **ISP** and **INTBASE**, are 21 bits wide. However, the five most significant bits of the **ISP** registers are always cleared. Therefore, program code, and the interrupt dispatch table, can reside anywhere in the CR16B 2 Mbytes address space. The interrupt stack must reside in the lowest 64 Kbytes of the address space.

D.2 MEMORY MAPS

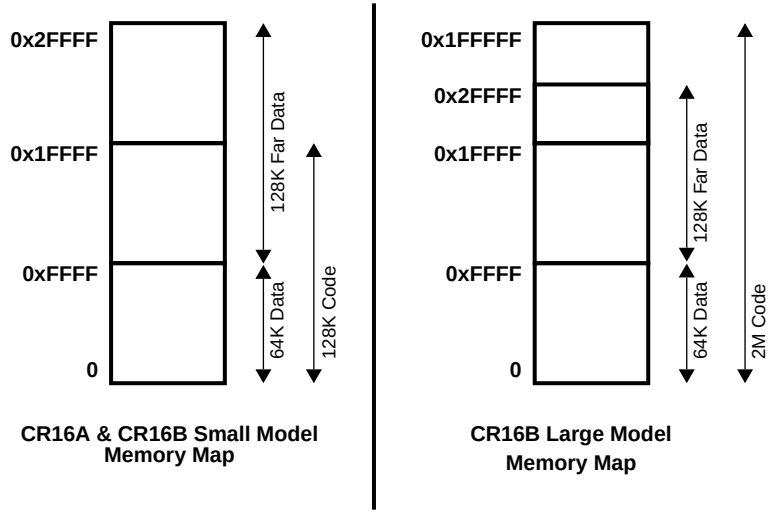


Figure D-2. Memory Maps

D.3 INSTRUCTION SET

The CR16B instruction set is a super-set of the CR16A instruction set. Thus, any program written for the CR16A can be assembled using the CR16B assembler. However, using the CR16B instruction set can reduce the code size of the and increase the performance of the program.

D.3.1 Bit Manipulation

The CR16A carries out all operations in registers, and only the **LOADi** and **STORi** instructions can access memory. Thus, CR16A applications use several instructions to manipulate a memory bit: loading the data from memory into a general-purpose register, modifying the specific bit with the **ORi** or **ANDi** instructions, and storing the result back into memory. Furthermore, this bit modification sequence is not atomic and there it is interruptable. If two asynchronous tasks may modify the same bit, one must protect the bit modification sequence by disabling interrupts during the modification.

The CR16B bit manipulation instructions, **SBIT**, **CBIT**, and **TBIT** set, clear, and test respectively a bit in memory. A memory bit is modified in a single instruction and this instruction is not interruptable.

Example Consider the following C code:

```
struct {
     int bit0:1;
     int bit1:1;
     int bit2:1;
     int bit3:1;
    } bit_flags;

bit_mask.bit3 = 1;
```

CR16A

```
di
loadb _bit_mask, r0
orb    $0x8, r0
storb r0, _bit_mask
ei
```

CR16B

```
sbitw 0x3,_bit_mask
```

D.3.2 `POP` / `PUSH` / `POPRET` Instructions

The **PUSH**, **POP** and **POPRET** instructions save and restore up to four sequential registers on/from the program stack.

PUSH

- Saves up to four consecutive general-purpose registers on the stack.
- Adjusts the value of the **SP** register accordingly.

POP

- Restores up to four consecutive general-purpose registers from the stack
- Adjusts the value of the **SP** register accordingly.

POPRET

- Restores up to four consecutive general-purpose registers from the stack
- Adjusts the value of the **SP** register accordingly.
- Jumps to the return address by copying the value of either the **RA** register (small model) or the **ERA-RA** register pair (large model) to the **PC**.

Note that the **PUSH**, **POP** and **POPREET** instructions assume that the **SP** general-purpose register is used as the program stack pointer.

These functions are very efficient for cases when a sequence of registers has to be saved on or restored from the program stack. The most common examples are entry and exit sequences of functions.

Example In the following example **func1** uses registers **r0**-**r7**.

CR16A

```
_func1::
add $-8, sp
storw r7, 0(sp)
storw r8, 2(sp)
storw r9, 4(sp)
storw r10, 6(sp)
          .
          .
loadw r7, 0(sp)
loadw r8, 2(sp)
loadw r9, 4(sp)
loadw r10, 6(sp)
add $8, sp
jump ra
```

CR16B

```
_func1::
push $4, r7
        .
        .
popret $4, r7
```

D.3.3 Multiply

The CR16A instruction set includes the following multiply instructions:

- **dest** (8-bits) = **src** (8-bits) * **dest** (8-bits)
- **dest** (16-bits) = **src** (16-bits) * **dest** (16-bits)

In addition to the CR16A multiply instructions, the CR16B includes the following multiply instructions:

- *sign* **dest** (16-bits) = *sign* **src** (8-bits) * *sign* **dest** (8-bits)
  Note: the application is responsible for zero extending the operand into 16-bit registers before the multiplication.
- *unsign* **dest** (16-bits) = *unsign* **src** (8-bits) * *unsign* **dest** (8-bits)
- *sign* **dest** (32-bits) = *sign* src (16-bits) * *sign* **dest** (16-bits)
- *unsign* **dest** (32-bits) = *unsign* **src** (16-bits) * *unsign* **dest** (16-bits)
  Note: restricted to a subset of the general-purpose registers (**r0**, **r1**, **r8** and **r9**).

Since there is no multiplication of a 16-bit register by a 16-bit register, either the compiler or the assembly user called a **__mulsi3** emulation library to perform the multiplication.

Example Consider the following C code:

```
short a, b;
long  l;

l = a * b;
```

CR16A

```
loadw     _a,r2
movw      r2,r3
ashuw     $-15,r3
loadw     _b,r4
movw      r4,r5
ashuw     $-15,r5
bal       ra,___mulsi3
storw     r0,_l
storw     r1,_l+2
```

CR16B

```
loadw     _a,r2
mulsw     r2,r3
```

D.3.4 MOVD

A CR16A application needs two **MOVW** instructions to load an immediate value of more than 16 bits. The CR16B can perform this operation in a single **MOVD** instruction, that loads an immediate of up to 21 bits into a pair of consecutive registers. This is especially useful for loading addresses of *far* pointers into register pairs.

D.3.5 Compare and Branch

Whenever a CR16A program needs to test the value of a variable stored in memory, and branch accordingly, it performs the following instructions:

- Loads the value of the variable into a general-purpose register (unless it was previously loaded).
- Compares the general-purpose register value with an immediate value.
- Conditionally branches to a destination address according to the results of the comparison.

In the CR16B it is possible to combine the last two steps into one, for a sub-set of general-purpose registers, and for a given immediate values of 0 and 1 (which are the most common immediate values in comparisons).

Example

```
extern int b, c;

void foo()
{
 int a;
 b = 5;
 if (c == 0)
     a = 1;
 if (a != 1)
     b = 3;
}
```

CR16A

```
_foo:
     movw      $_b,r1
     movw      $5,r0
     storw     r0,0(r1)
     loadw     _c,r0
     cmpw      $0,r0
     bne       .L2
     movw      $1,r2
.L2:
     cmpw      $1,r2
     beq       .L3
     movw      $3,r0
     storw     r0,0(r1)
.L3:
     jump      ra
```

CR16B

```
_foo:
     movw      $_b,r2
     movw      $5,r3
     storw     r2,0(r3)
     loadw     _c,r0
     bne0      r0,.L2
     movw      $1,  r1
.L2:
     beq1      r1,.L3
     movw      $3,r3
     storw     r3,0(r1)
.L3:
     jump      ra
```

D.3.6 Store Small Immediate

In the CR16A, two steps are required to store an immediate value into memory:

- Load the immediate value into a general-purpose register using a **MOVi** instruction.
- Store the register into memory using a **STORi** instruction.

In the CR16B it is possible to store small immediate values directly into memory, without loading the immediate value into an intermediate general-purpose register. This is done with the **STORi** instruction, which now supports a 4-bit immediate value in the source operand. The destination operand is either an 18-bit absolute address, or a register-relative operand. Note that in the latter case only **r0**, **r1**, **r8** and **r9** are supported as base registers.

Example The following example shows how to improve the code shown in Section D.3.5.

```
extern int x, *p;

foo()
{
  x = 0;
 *p = 1;
}
```

CR16A

```
_foo:
          movw     $0,r0
          storw    r0,_x
          loadw    _p,r1
          movw     $1,r0
         storw    r0,0(r1)
          jump     ra
```

CR16B

```
_foo:
      storw    $0,_x
      loadw    _p,r0
      storw    $1,0(r0)
      jump     ra
```

D.4 EIWAIT

The new CR16B **EIWAIT** instruction first enables the interrupt and then waits, as opposed to the **WAIT** instruction, which does not enable interrupts.

INDEX

A

absolute addressing mode 2-17
acknowledge
  exception 3-6
ADDCi instructions 6-5
ADDi instructions 6-4
addition
  integer instructions, **ADD[U]i** 6-4
  integer with carry instructions, **ADDCi** 6-5
  with carry 2-9
address
  compare 4-1, 4-3, 4-4
  registers, dedicated 2-7
address-compare match, write
  bit, DSR.BWR 4-4
addressing mode
  absolute 2-17
  immediate 2-16
  in instructions 6-1
  register 2-15
  relative 2-16
ADDUi instructions 6-4
ANDi instructions 6-6
arithmetic
  shift instructions, **ASHUi** 6-7
ASHUi instructions 6-7

B

BAL instruction 6-12
Bcond instructions 6-8
bitwise logical
  AND instructions, **ANDi** 6-6
  OR instructions, **ORi** 6-44
boolean data type 2-2
boolean, instructions to save condition as, **Scond** 6-51
borrow, see also carry 2-9
BPC, PC bit in DSR 4-4
BPC, PC match bit in DSR 4-4
BPT trap 3-3, 3-12
BR instruction 6-14
branch
  and link instruction, **BAL** 6-12
  unconditional, instruction, **BR** 6-14
BRD, compare address bit in DSR 4-4
breakpoint
  generation 4-1
  trap, BPT 3-3, 3-12
BWR, write, address-compare match bit in DSR 4-4
byte order
  for data references 2-14

C

C, carry bit in PSR 2-9
CAR register 2-11, 4-1, 4-2, 4-3
carry bit, PSR.C 2-9
CBE0-3, compare-address enable bits in DCR 4-5
clock
  cycle A-1
CLRBi instructions 6-16
CMP0BCondi instructions 6-19
CMPi instructions 6-18
compare address
  for debugging 4-1, 4-3
compare address bit in DSR
  BRD 4-4
compare-address
  bits in DCR 4-5
  PC match enable bits 4-5
comparison
  integer instructions, **CMP0BCondi** 6-19
  integer instructions, **CMP1BCondi** 6-20
  integer instructions, **CMPi** 6-18
  operations 2-9
cond, condition code 6-51
conditional instructions
  branch, **Bcond** 6-8
  jump, **Jcond** 6-25
  save, **Scond** 6-51
configuration register, see also CFG 2-10
convert
  sign integer to word, **MOVXB** 6-37
  unsigned integer to unsigned double-word, **MOVZi** 6-38
  unsigned integer to unsigned word, **MOVZB** 6-38
CRD, compare address read enable bit in DCR 4-5
CWR, compare address write enable bit in DCR 4-4, 4-5
cycle, in instruction execution timing A-1

D

data
  length attribute specifier in instructions 6-1
  organization 2-12

references, byte order 2-14
types 2-1
DBG trap
and exception service procedures 3-9
DCR register 2-11, 4-1, 4-4
debug
control register, DCR 2-11, 4-4
dedicated address registers 2-7
delays during instruction execution A-1
DEN, address-compare and PC match enable bit in DCR 4-5
DI instruction 6-21
DISABLE instruction 2-10
dispatch table, IDT
in SF architecture 3-1
see also IDT 3-1
division by zero
trap, DVZ 3-3
DSR register 2-11, 4-1
DVZ trap 3-3

E

E, local maskable interrupt enable bit in PSR 2-10
EI instruction 6-22
EIWAIT instruction 6-23
ENABLE instruction 2-10
encoding, instruction set B-1
exception
acknowledge 3-6, 3-8
defined 3-1
handler 3-1
instruction, **EXCP** 6-24
priority 3-10
processing 3-4
processing table 3-9
processing, flowchart 3-11
return instruction, **RETX** 6-48
service procedure 3-9
EXCP instruction
and serialized instructions 4-9
EXCP instruction 3-10, 6-24
executing-instructions operating state 4-7
execution
program suspension instruction, **EIWAIT** 6-23
program suspension instruction, **WAIT** 6-63
timing for instructions A-1

F

F
flag bit of PSR 2-9, 6-61
fetch
stage in integer pipeline, IF 2-8
flag
bit, PSR.F 2-9, 6-61
FLG trap 3-3

G

general purpose registers 2-7

H

handler
exception 3-1

I

I, maskable interrupt enable bit of PSR 2-10
ICU, interrupt control unit 4-8
ID, stage in integer pipeline A-1, A-2
IDT, interrupt dispatch table 3-1, 4-8
IF
stage in integer pipeline 2-8
immediate
addressing mode 2-16
instruction
decoding, stage in integer pipeline, ID A-1, A-2
dependency 4-9
endings 3-4
execution timing A-1
format 6-1
latency, defined A-1
latency, in example A-3
parallel execution 4-8
pipeline execution 4-8
serial execution 4-9
set, encoding B-1
set, summary 2-2
suspended, completion 3-9
throughput, defined A-1
tracing 4-1
In-System Emulator interrupt, see ISE interrupt
INTBASE register 2-8
integer
addition instructions, **ADD[U]i** 6-4
addition with carry instructions, **ADDCi** 6-5
arithmetic shift instructions, **ASHUi** 6-7
comparison instructions, **CMP0BCondi** 6-19
comparison instructions, **CMP1BCondi** 6-20
comparison instructions, **CMPi** 6-18
convert to unsigned 6-38
data type 2-1
load instructions, **CLRBi** 6-16
load instructions, **LOADi** 6-30
load instructions, **SETBi** 6-49
logical shift integer instructions, **LSHi** 6-35

move instructions, **MOVi** 6-36
multiplication instructions, **MULi** 6-39
multiplication instructions, **SMULB** 6-40
multiplication instructions, **SMULW** 6-41
multiplication instructions, **UMULW** 6-42
pipeline organization A-2
sizes 2-1
store instructions, **STORi** 6-55
subtract with carry instructions, **SUBCi** 6-60
subtraction instruction, **SUBi** 6-59
internal register 2-6
interrupt
defined 3-1
dispatch table, IDT 3-1
maskable 2-10
maskable, **DI** instruction 6-21
maskable, **EI** instruction 6-22
non-maskable 2-10, 3-2
priority 3-10
stack pointer, see also ISP register 2-8
stack, and **RETX** instruction 6-48
stack, description 2-14
stack, during exception 3-2, 3-10
vector, see also, dispatch table, INTBASE 6-24
wait for interrupt instruction, **EIWAIT** 6-23
wait for interrupt instruction, **WAIT** 6-63
ISE interrupt 3-2
ISE support 4-1
ISP register 2-8

J

JAL instruction 6-27
Jcond instructions 6-25
jump
conditional, instructions, **Jcond** 6-25
jump and link instruction, **JUMP** 6-29
jump and link insturction, **JAL** 6-27
JUMP instruction 6-29

L

L, low flag of PSR 2-9
latency, instruction A-1
link after branch instruction, **BAL** 6-12
load
integer instructions, **CLRBi** 6-16
integer instructions, **LOADi** 6-30
integer instructions, **SETBi** 6-49
processor register instruction, **LPR** 6-33
LOADi instructions 6-30
logical
AND instructions, **ANDi** 6-6
exclusive OR instructions, **XORi** 6-64
OR instructions, **ORi** 6-44
shift integer instructions, **LSHi** 6-35
low flag, PSR.L 2-9
LPR instruction
and PSR.P bit 4-2
and serialized instructions 4-9
LPR instruction
accessing DCR, DSR 4-1
description 6-33
LSHi instructions 6-35

M

maskable
interrupt enable bit, PSR.E 2-10
maskable interrupt 3-2
maskable interrupt disable instruction, **DI** 6-21
maskable interrupt enable bit, PSR.I 2-10
maskable interrupt enable instruction, **EI** 6-22
memory
organization 2-12
references using **LOAD** and **STORE** 2-14
model, programming 2-1
move
integer instructions, **MOVi** 6-36
MOVi instructions 6-36
MOVXB instruction 6-37
MOVZB instruction 6-38
MULi instructions 6-39
multiplication
integer instructions, **MULi** 6-39
integer instructions, **SMULB** 6-40
integer instructions, **SMULW** 6-41
integer instructions, **UMULW** 6-42

N

N, negative bit in PSR 2-10
negative bit, PSR.N 2-10
no operation instruction, **NOP** 6-43
non-maskable interrupt 3-2
NOP instruction 6-43

O

operand
access class and length in instructions 6-1
in instructions 6-1
OR logical
exclusive, instructions, **XORi** 6-64
order, byte, for data references 2-14
ORi instructions 6-44

P

P, trace trap pending bit in PSR 2-10, 4-1
parallel processing
  in pipeline 4-8
PC match
  and compare-address enable bits 4-5
PC match bit, DSR.BPC 4-4
PC register
  and exceptions 3-2
  bit, DSR.BPC 4-4
  match 4-1, 4-2
  match enable bits in DCR 4-5
pipeline
  organization, integer A-2
pipelined instruction execution 4-8
POPrt instructions 6-45
priority, exception 3-10
processing-an-exception operating state 4-7
processor
  registers and load instruction, **LPR** 6-33
  status register, see also PSR 2-8, 3-2
program
  execution time A-1
  modes 2-1
  stack 2-14
PSR register
  and CMPi instructions 6-18
  and **DI** instruction 6-21
  and exceptions 3-2
  description 2-8
PUSH instructions 6-47

R

R0, R1 registers 2-9
references to memory 2-14
register
  addressing mode 2-15
  configuration, see also CFG 2-10
  dedicated address 2-7
  general purpose 2-7
  internal 2-6
  processor, and store instruction, **SPR** 6-53
relative addressing mode 2-16
reset 3-13, 4-7
resume execution after **EIWAIT** 6-23
resume execution after **WAIT** 6-63
return
  from exception instruction, **RETX** 6-48
RETX instruction
  after exceptions 3-2
  and serialized instructions 4-9
  tracing 4-2
RETX instruction
  description 6-48
  in exception service procedure 3-10
$\overline{\text{RST}}$ signal 3-13

S

save, on condition instructions, **Scond** 6-51
Scond instructions 6-51
SETBi instructions 6-49
shift
  arithmetic, instructions, **ASHUi** 6-7
  logical, integer instructions, **LSHi** 6-35
signed integer data type 2-1
SMULB instructions 6-40
SMULW instructions 6-41
SP
  general purpose register 2-7
SPR instruction
  accessing DCR, DSR 4-1
  description 6-53
stack
  interrupt and program 2-14
  interrupt, during exception 3-2, 3-10
  interrupt, in **RETX** instruction 6-48
store
  integer instructions, **STORi**i 6-55
  processor register instruction, **SPR** 6-53
STORi instructions 6-55
SUBCi instructions 6-60
SUBi instructions 6-59
subtraction
  integer instruction, **SUBi** 6-59
  with carry 2-9
  with carry, integer instructions, **SUBCi** 6-60
supervisor
  call trap, SVC 3-3
suspend execution instruction, **EIWAIT** 6-23
suspend execution instruction, **WAIT** 6-63
SVC trap 3-3

T

T, trace bit in PSR 2-9, 4-1
TBIT instruction 2-14, 6-61
test bit instruction, **TBIT** 6-61
throughput, instruction
  defined A-1
timing
  instruction execution A-1
trace
  bit, PSR.T 2-9, 4-1
  trap pending bit, PSR.P 2-10, 4-1
  trap TRC, description 3-3
tracing
  instructions 4-1
  program 2-9
trap
  defined 3-1
  list and descriptiohns 3-3
  table with vecor for each type 6-24
  trace, TRC 2-10

TRC trap
description 3-3
in exception service procedure 3-9
pending bit, PSR.P 2-10

U

UMULW instructions 6-42
unconditional branch instruction, **BR** 6-14
UND trap 3-3
definition 3-10
undefined
instruction trap, UND 3-3
undefined instruction
trap, UND 3-10
unsigned integer data type 2-1

V

vector
interrupt table 6-24

W

WAIT instruction 4-8, 6-63
waiting-for-an-interrupt operating state 4-7

X

XORi instructions 6-64

Z

Z, zero bit in PSR 2-9
zero
bit, PSR.Z 2-9