CompactRISC™

CR16C

Programmer's Reference Manual

Part Number: 424521772-101

March 2002

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
1.0	December 2000	First release.
1.1	February 2001	Changed BAL/Bcond to ±16 MByte. Minor clarificatons/corrections.
3.1	March 2002	Product release for CR16B/C.

PREFACE

This Programmer's Reference Manual presents the programming model for the CompactRISC CR16C microprocessor core. The key to system programming, and a full understanding of the characteristics and capabilities 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.

This page is intentionally blank.

CONTENTS

Chapter 1
INTRODUCTION

1.1 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. The CompactRISC architecture 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 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.

1.2 CR16C 16-BIT CompactRISC PROCESSOR CORE

The CR16C is a third-generation 16-bit CompactRISC processor core. It is assembly-level compatible with its predecessors, the CompactRISC CR16A and CR16B, and provides expanded options for system designers. The new implementation provides:

1. Address Space
   - Expanded linear address space of 16 Mbytes for program code and data memory
   - User, supervisor and interrupt stack pointers covering the full address range

2. Instruction Set Enhancements

   – Double-word support for most instructions improve optimizations for data and code access above the first 64K.
     - load and store
     - move and movex/z
     - arithmetic (compare, add, subtract)
     - logic (AND, OR, XOR)
     - shifts (arithmetic and logical)

   – Expanded push/pop instructions allow up to eight registers with a separate bit to determine if the return address should also be pushed/popped to/from the stack

   – expanded load/store multiple instructions to allow up to 8 registers.

   – optional mac instruction

3. Addressing Modes Enhancements

   – Index addressing mode for better support of relocatable code

   – Register pair relative addressing mode, with efficient instruction encoding for all memory access instructions, improves optimizations for data and code access above the first 64K.

4. System Features

   – User/supervisor mode

   – Illegal address trap (address out of range)

   – Cache support

   – Enhanced debug features, with up to eight hardware breakpoints

5. Speed Improvements

   – Enhanced pipelining of data transfers

   – Faster/deterministic multiply (single cycle 8*8)

To ensure a seamless transition for existing CompactRISC users, the CR16C provides a configuration bit, CFG.SR, that permits exclusive use of only small registers. This mode is backward compatible with the large programing model of the CR16B. The small programing model of the CR16B, which is backward compatible with the CR16A, is no longer supported.

1.3 THE COMPACTRISC ARCHITECTURE

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

For example:

- The CR16C executes an optimized instruction set with up to 33 internal registers grouped in 16 general-purpose registers, four dedicated address registers, a processor status register, a configuration register and up to 11 debug-control registers.
- The CR16C 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 CR16C 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 CR16C performs fast multiply operations using a 16-bit by 8-bit hardware multiplier.

In general, the CompactRISC architecture supports little-endian memory addressing. This means that the byte order in the CR16C is from the least significant byte (LSB) to the most significant byte (MSB).

1.4 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 CR16C may be encoded in 2, 4 or 6 bytes, but basic instructions are only 2 bytes long. This permits optimized instruction processing by the instruction decoder, and results in a smaller code size. Code generated for the CR16C is comparable to CISC code size, or typically 50 percent smaller than code generated for leading 32-bit RISC CPUs. Another advantage is the ability to generate performance with lower pin-counts or lower bandwidth buses, again a trait of an embedded system.

32-bit RISC processors store registers and addresses on the stack as 32-bit entities. The CR16C 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.

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

- Reduced costs
- Many more system elements integrated with on-chip memory
- Fewer pins to access minimum-sized off-chip memory
- Larger amounts of on-chip memory than similar processors, at the same cost.

1.5 SCALABLE ARCHITECTURE FROM 8 TO 64 BITS

The architectural features described above make the CompactRISC technology ideal for the next generation of embedded systems. One additional design decision opened the door for CompactRISC technology to be effectively used from low-end to high-end embedded systems: the CompactRISC architecture is 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 using one CompactRISC core can easily migrate to another core, due to the high similarity in both the architecture and the development tools.

1.6 MODULAR EXTENSIONS

The CompactRISC technology was designed to be easily extended. 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 a 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.

1.7 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 with enhanced simulation capabilities. In addition, an integrated, multiple core, graphical debugger supports debugging of multiple CompactRISC/DSP cores on a single die.

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.

The CR16C supports a NEXUS class 1 compliant on-chip debug module. With this interface, every debugger that uses the industry standard NEXUS API can be used for CR16C-based System-on-a-Chip (SOC) designs.

The CompactRISC architecture is also well supported with Real-Time Operating Systems (RTOS) from third party vendors.

As a package, these tools simplify the task of designing and developing advanced embedded systems in high level languages such as ANSI-C.

This page is intentionally blank.

Chapter 2
PROGRAMMING MODEL

This chapter describes the CR16C register set and the instruction set. The CR16C supports up to 16 Mbytes of program and data space. Five addressing modes are supported: register, immediate, absolute, relative and indexed. Refer to Section 2.5 for an overview of the memory areas with most efficient data accesses.

2.1 COMPATIBILITY WITH CR16A AND CR16B

The CR16C is backward-compatible with the CR16A and CR16B. The CR16C maintains assembly level compatibility, not binary compatibility, with the CR16B large model. Code that was developed for the CR16B large model runs on the CR16C after being reassembled for the CR16C. Appendix D provides a summary of the differences between the CR16A/B and the CR16C programming models.

2.2 DATA TYPES

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

Integer data type

The integer data type is used to represent integers. Integers may be signed or unsigned. Three integer sizes are supported: 8-bit (1 byte), 16-bit (1 word) and 32-bit (1 double-word). Signed integers are represented as binary 2's complement numbers, with values in the range -2^7 to 2^7-1, -2^{15} to $2^{15}-1$ and -2^{31} to $2^{31}-1$, respectively. Unsigned numbers have values in the range 0 to 2^8-1, 0 to $2^{16}-1$ and 0 to $2^{32}-1$, respectively.

Boolean data type

The Boolean data type is represented as an integer (either a byte or a word). The value of its least significant bit represents one of two logical values: true (integer 1) or false (integer 0). Bits other than the least significant bit are not interpreted.

2.3 REGISTER SET

This section describes the register set of the CR16C. The format of each register is illustrated and described in detail. Bits/Registers specified as “reserved” must be written as 0, and return undefined results when read. Non-implemented registers are read as zero.

The internal registers of the CR16C are grouped by function:

- 16 general-purpose registers including four double registers:
  - Two available for use in index addressing mode
  - One used as a return address pointer
  - One used as the stack pointer
- Four dedicated address registers
- One Processor Status Register
- One Configuration Register
- Five debug registers supporting two debug channels.
- Optionally, additional sets of four debug registers for each pair of auxiliary debug channels.

Figure 2-1 shows the internal registers of the CR16C.

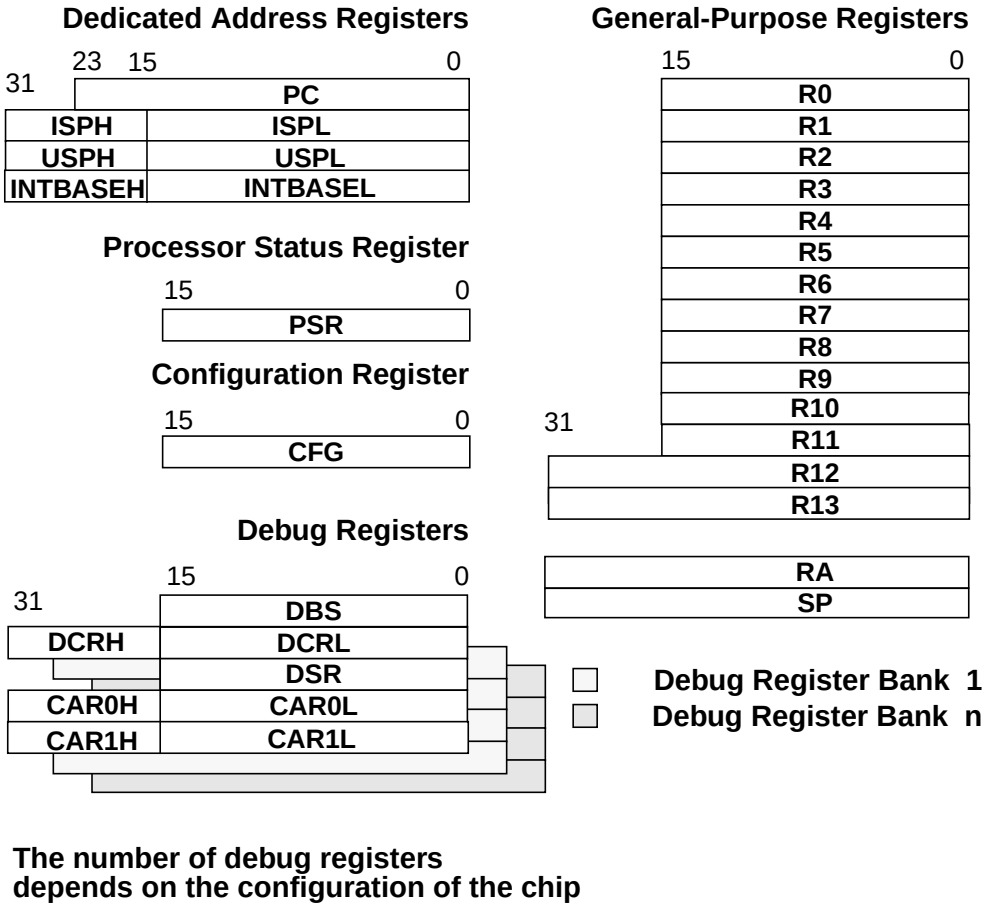


Figure 2-1. CR16C Internal Registers

2.3.1 General-Purpose Registers

The CompactRISC cores feature 16 general-purpose registers. These registers are used individually as 16-bit operands or as register pairs for operations with 32-bit operands or on addresses greater than 16 bits.

- Registers are defined as R0 through R13, RA and SP.
- Register pairs are defined based on the setting of the Short Register bit in the Configuration Register (CFG.SR). When CFG.SR is set, register pairs are defined as in the CR16A/B:
  (R1,R0), (R2,R1) ... (R11,R10), (R12_L, R11), (R13_L, R12_L),
  (R14_L, R13_L) and SP. (R14_L, R13_L) is the same as (RA,ERA).
- When CFG.SR is not set, register pairs are defined as follows:
  (R1,R0), (R2,R1) ... (R11,R10), (R12_L, R11),
  R12, R13, RA, SP.
  R12, R13, RA and SP are double-word registers for direct storage of addresses greater than 16 bits.

With the recommended calling convention for the architecture, some of these registers are assigned special hardware and software functions. Registers R0 to R13 are used for general purposes, such as holding variables, addresses or index values. The SP Register is used as a pointer to the program run-time stack. The RA Register is used as a return address from subroutines. R12 and R13 are available for use as a base index address in the index addressing mode.

If a general-purpose register is specified by an operation that is 8 bits long, only the lower part of the register is used; the higher part is not referenced, or modified. Similarly, for word operations on register pairs, only the lower register is used. The upper register is not referenced or modified.

2.3.2 Dedicated Address Registers

This section describes the four, double-word wide, dedicated address registers that the CR16C uses to implement specific address functions.

Program Counter (PC) Register

The value in the PC Register points to the first byte of the instruction currently being executed. CR16C instructions are aligned to even addresses, thus the least significant bit of the PC is always 0. At reset, the PC is initialized to 0 or an alternative predetermined value. The value of the PC prior to reset is saved in the (R1,R0) general-purpose register pair.

Interrupt Stack Pointer (ISP)

The ISP 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. This stack pointer is accessible as the ISP Processor Register for initialization. The interrupt stack can exist in the entire address range. The ISP cannot be used for any purpose other than the automatic storage of registers on the interrupt stack during an exception, and the restoration of these registers during a RETX. The interrupt stack grows downward in memory. See Section for additional details. The most significant 8 bits of ISP and the least significant bit of ISP are forced to 0.

User Stack Pointer (USP)

The USP points to the lowest address of the last item stored on the user stack. This stack is used when the program stack is used for supervisor and user level processes. This stack also grows downward in memory. See “The program stack” on page 2-14 for more information. If the USP points to an illegal address (any address greater than 0x00FF_FFFF) the execution of a stack modifying instruction (PUSH or POP) an IAD trap.

Interrupt Base (INTBASE) Register

The INTBASE Register holds the address of the dispatch table for interrupts and traps. The dispatch table can be located anywhere in the supported addresses range. The most significant 8 bits of INTBASE and the least significant bit of INTBASE are forced to 0. See Chapter 3 for more information.

2.3.3 The Processor Status Register

The 16-bit wide Processor Status Register (PSR) holds status information and selects operating modes for the CR16C. See Figure 2-2 for the PSR bit assignment.

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

Figure 2-2. PSR Register

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 R2 general-purpose register.

Bits Z, C, L, N, and F in the PSR have a dedicated condition code in conditional branch instructions. Any conditional branch instruction can cause a branch in program execution, based on the value of one or more of these PSR bits. 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 Section 5.1 on page 5-1 for details.

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

Bit C 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.

Bit T The Trace bit causes program tracing. When the T bit is set to 1, a Trace (TRC) trap is executed after every instruction. Refer to Section 4.1.1 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). When a hardware debug module is present in the system, the value of the T bit is ignored.

Bit L 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 Section 5.1 on page 5-1 for details.

Bit U The User Mode bit is set to indicate that the User Stack Pointer is being used as the Stack Pointer. This bit can only be set by the Jump USR instruction. This bit is cleared before any exception processing, such as an interrupt or a trap. When this bit is clear, the supervisor stack pointer is used. The USP Register is accessed as a processor register when the User Mode bit is not set. When the User Mode bit is set, Load Processor Register LPR) operations are not permitted. The value of this bit is output on signal SFUSR from the core.

Bit F 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). It is also set, or cleared, as a result of a Test, Set or Clear bit instruction.

Bit Z 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 Section 5.1 on page 5-1 for details.

Bit N 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 Section 5.1 on page 5-1 for details.

Bit E The local maskable interrupt Enable bit affects the state of maskable interrupts. When this bit and the PSR.I bits are 1, all maskable interrupts are accepted. When this bit is 0, only the non-maskable interrupt is accepted. See Section 3.2 on page 3-2.

There are two dedicated instructions that set and clear the E bit: the Enable Interrupts instruction (**EI**) sets it to 1; the Disable Interrupts instruction (**DI**) clears it to 0. 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 Section 3.2 on page 3-2.

Bit P 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 is cleared when no TRC trap is pending. It is set when a TRC trap is pending. See Sections 3.4.4 on page 3-9 and 4.1.1 on page 4-1 for more information. When a hardware debug module is present in the system the value of the P bit is ignored.

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

2.3.4 The Configuration Register

The Configuration Register (CFG) is used to enable or disable various operating modes and to control optional on-chip caches.Figure 2-3 shows the CFG Register bit assignment.

15-10	9	8	7	6	5	4	3	2	1	0
reserved	SR	ED	0	0	LIC	IC	LDC	DC	0	0

Figure 2-3. Configuration Register

All CFG bits are cleared on reset. The CFG control bits are described in detail below.

Bit DC If the Data Cache bit is set to 1, the data cache is accessible for data read and write operations. If the DC bit is cleared to 0, the data cache is disabled.

Bit LDC If the Lock Data Cache line bit is set to 1, a missing line, which cannot be replaced, is locked into the cache after it is placed. If LDC is cleared to 0, the line can be replaced.

Bit IC If the Instruction Cache bit is set to 1, the instruction cache is accessible for instruction fetches. If the IC bit is cleared to 0, the instruction cache is disabled.

Bit LIC If the Lock Instruction Cache line bit is set to 1, a missing instruction is locked into the instruction cache after it is placed. It is not replaced as long as LIC remains 1.

Bit ED The Extended Dispatch bit determines the size of an entry in the Interrupt Dispatch Table. Each entry holds the address of the appropriate exception handler. When ED is set, the Interrupt Dispatch Table contains 32-bit elements, each occupying two adjacent words. When ED is cleared, the Interrupt Dispatch Table contains 16-bit elements. This implies that when ED is clear, all exception handlers must start in the first 128K of the address space. On reset, the bit is cleared. The location of the Interrupt Dispatch Table is determined by INTBASE, independent of the setting of the ED bit.

Bit SR The Short Register bit is set to enable the register pairing used to maintain compatibility with code developed for the CR16B large model. As opposed to using the extended versions of R12, R13 and R14, only the lower 16 bits of these registers are used, and are paired together as register pairs for double operations. The (R14, R13) register pair is used as the extended RA. In addition, when this bit is set, address displacements relative to a single register are supported with offsets of 0 and 14 bits in place of the index addressing with these displacements. See “Index mode” on page 2-16 for additional details.

The Debug RegistersDebug Base Register (DBS)

The Debug Base Register specifies which set of debug registers is mapped into the debug register space (CAR0/1, DCR and DSR). In addition, a global debug status is available via the DBS. For more information, see Section 4.1.1 on page 4-1.

Compare Address Registers (CAR0/1)

The Compare Address Registers (CAR0 and CAR1) contain the address to be used to generate a breakpoint on an address or data. For more information, see Section 4.1.1 on page 4-1.

Debug Control Register (DCR)

The Debug Control Register (DCR) holds the debug control bits. For more information, see Section 4.1.1 on page 4-1.

Debug Status Register (DSR)

The Debug Status Register (DSR) holds the debug status bits. For more information, see Section 4.1.1 on page 4-1.

2.4 INSTRUCTION SET

The following table summarizes the CR16C instruction set. Chapter 5 and Appendix B describe each instruction in detail. Also refer to "Instructions Table Glossary" on page 2-11.

Mnemonic	Operands	Description
MOVES		
MOVi	Rsrc/imm, Rdest	Move
MOVXB	Rsrc, Rdest	Move with sign extension
MOVZB	Rsrc, Rdest	Move with zero extension
MOVXW	Rsrc, RPdest	Move with sign extension
MOVZW	Rsrc, RPdest	Move with zero extension
MOVD	imm, RPdest RPsrc, RPdest	Move immediate to register-pair Move between register-pairs
INTEGER ARITHMETIC		
ADD[U]i	Rsrc/imm, Rdest	Add
ADDCi	Rsrc/imm, Rdest	Add with carry
ADDD	RPsrc/Imm,RPdest	Add with RP or immediate.
MACQW	Rsrc1 Rsrc2, RPdest	Multiply signed Q15: RPdest := RPdest + Rsrc1 * Rsrc2
MACSW	Rsrc1 Rsrc2, RPdest	Multiply signed and add result: RPdest := RPdest + Rsrc1 * Rsrc2
MACUW	Rsrc1 Rsrc2, RPdest	Multiply unsigned and add result: RPdest := RPdest + Rsrc1 * Rsrc2
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, RPdest	Multiply: RPdest := RPdest(16) * Rsrc(16)
MULUW	Rsrc, RPdest	Multiply: RPdest := RPdest(16) * Rsrc(16);
SUBi	Rsrc/imm, Rdest	Subtract: (Rdest := Rdest − Rsrc)
SUBD	RPsrc/imm, RPdest	Subtract: (RPdest := RPdest − RPsrc/imm)
SUBCi	Rsrc/imm, Rdest	Subtract with carry: (Rdest := Rdest − Rsrc)

Mnemonic	Operands	Description
INTEGER COMPARISON		
CMPi	Rsrc/imm, Rdest	Compare Rdest − Rsrc
CMPD	RPsrc/imm, RPdest	Compare RPdest-RPsrc
BEQ0i	Rsrc, disp	Compare Rsrc to 0 and branch if EQUAL
BNE0i	Rsrc, disp	Compare Rsrc to 0 and branch if NOT-EQUAL
LOGICAL AND BOOLEAN		
ANDi	Rsrc/imm, Rdest	Logical AND: Rdest := Rdest & Rsrc/Imm
ANDD	RPsrc/imm, RPdest	Logical AND: RPdest := RPsrc & RPsrc/Imm
ORi	Rsrc/imm, Rdest	Logical OR: Rdest := Rdest | Rsrc/Imm
ORD	RPsrc/imm, RPdest	Logical OR: Rdest := RPdest | RPsrc/Imm
Scond	Rdest	Save condition code as boolean
XORi	Rsrc/imm, Rdest	Logical exclusive OR: Rdest := Rdest ^ Rsrc/Imm
XORD	RPsrc/imm, RPdest	Logical exclusive OR: Rdest := RPdest ^ RPsrc/Imm
SHIFTS		
ASHUi	Rsrc/imm, Rdest	Arithmetic left/right shift
ASHUD	Rsrc/imm, RPdest	Arithmetic left/right shift
LSHi	Rsrc/imm, Rdest	Logical left/right shift
LSHD	Rsrc/imm, RPdest	Logical left/right shift
BIT OPERATIONS		
SBITi	Iposition, disp(Rbase) Iposition, disp(RPbase) Iposition, (Rindex)disp(RPbasex) Iposition, abs Iposition, (Rindex)abs	Set a bit in memory
CBITi	Iposition, disp(Rbase) Iposition, disp(RPbase) Iposition, (Rindex)disp(RPbasex) Iposition, abs Iposition, (Rindex)abs	Clear a bit in memory
TBIT TBITi	Rposition/imm, Rsrc Iposition, disp(Rbase) Iposition, disp(RPbase) Iposition, (Rindex)disp(RPbasex) Iposition, abs Iposition, (Rindex)abs	Test a bit in a register Test a bit in memory
PROCESSOR REGISTER MANIPULATION		
LPR	Rsrc, Rproc	Load processor register
LPRD	RPsrc, Rprocd	Load double processor register
SPR	Rproc, Rdest	Store processor register
SPRD	Rprocd, RPdest	Store double processor register

Mnemonic	Operands	Description
JUMPS AND LINKAGE		
Bcond	disp9 disp17 disp25	Conditional branch
BAL	RPlink, disp25	Branch and link
BR	disp9 disp17 disp25	Branch
EXCP	vector	Trap (vector)
Jcond	RPtarget	Conditional Jump to a large address
JAL	RA, RPtarget, RPlink,RPtarget	Jump and link to a large address
JUMP	RPtarget	Jump
JUSR	RPtarget	Jump and set PSR.U
RETX		Return from exception
PUSH	imm,Rsrc, RA	Push “imm” number of registers on user stack, starting with Rsrc and possibly including RA
POP	imm, Rdest, RA	Restore “imm” number of registers from user stack, starting with Rdest and possibly including RA
POPRET	imm, Rdest, RA	Restore registers (similar to POP) and JUMP RA
LOAD AND STORE		
LOADi	disp(Rbase), Rdest	Load (register relative)
	abs,Rdest	Load (absolute)
	(Rindex)abs, Rdest	Load (absolute index relative)
	(Rindex)disp(RPbasex), Rdest	Load (register relative index)
	disp(RPbase), Rdest	Load (register pair relative)
LOADD	disp(Rbase), RPdest	Load (register relative)
	abs, RPdest	Load (absolute)
	(Rindex)abs, RPdest	Load (absolute index relative)
	(Rindex)disp(RPbasex), RPdest	Load (register pair index relative)
	disp(RPbase), RPdest	Load (register pair relative)
STORi	Rsrc, disp(Rbase)	Store (register relative)
	Rsrc, disp(RPbase)	Store (register pair relative)
	Rsrc, abs	Store (absolute)
	Rsrc, (Rindex)disp(RPbasex)	Store (register pair relative index)
	Rsrc, (Rindex)abs	Store (absolute index)
STORD	RPsrc, disp(Rbase)	Store (register relative)
	RPsrc, disp(RPbase)	Store (register pair relative)
	RPsrc, abs	Store (absolute)
	RPsrc, (Rindex)disp(RPbasex)	Store (register pair index relative)
	RPsrc, (Rindex)abs	Store (absolute index relative)
STOR IMM	imm4, disp(Rbase) imm4, disp(RPbase) imm4, (Rindex)disp(RPbasex) imm4, abs imm4, (Rindex)abs	Store 4 bit immediate value in memory
LOADM	imm3	Load 1 to 8 registers (R2-R5, R8-R11) from memory starting at (R0)
LOADMP	imm3	Load 1 to 8 registers (R2-R5, R8-R11) from memory starting at (R1, R0)
STORM	imm3	Store 1 to 8 registers (R2-R5, R8-R11) to memory starting at (R1)
STORMP	imm3	Store 1 to 8 registers (R2-R5, R8-R11) to memory starting at (R7, R6)

Mnemonic	Operands	Description
MISCELLANEOUS		
CINV	[d, i, u]	Cache Invalidate
DI		Disable maskable interrupts
EI		Enable maskable interrupts
EIWAIT		Enable maskable interrupts and wait for interrupt
NOP		No operation
WAIT		Wait for interrupt

Instructions Table Glossary

R??? - Any general-purpose register (R0,R1, R2, R3...R12_L, R13_L, RA_L,SP)
Rsrc - Source register R???
Rdest - Destination register R???
Rbase - Base register for relative addressing R???
Rproc - Processor registers (CAR0/1, DCRL/H. DSRL/H, PSR, ISPL/H, etc.)
Rprocd- Processor registers Double (CAR0/1, DCR. DSR, PSR, ISP, etc.)
Rlink - Link register R??? holding the address of the next sequential address (return address).
Rtarget- Target register R???. The register holds a code address
Rindex- R12 or R13 used as an index register holding a base address.
Rposition- Bit position register R??? - only the lower 4-bits are used as bit position in TBIT
RP???- Any general-purpose register pair
when SR= 0: ((R1,R0) (R2,R1)...(R11,R10), (R12_L,R11), R12, R13,RA,SP.)
when SR= 1: ((R1,R0) (R2,R1)...(R11,R10), (R12_L,R11),
(R13_L,R12_L),(RA_L, R13_L),(SP_L,RA_L),SP.)
RPsrc - Source register pair RP???
RPdest- Destination register pair RP???
RPbase- Base register pair for relative addressing RP???
RPbasex- Base register pair for relative addressing only:
(R1,R0), (R3,R2), (R5,R4), (R7,R6), (R9,R8), (R11,R10), (R4,R3), (R6,R5)
RPlink - Link register pair RP???. The link register holds the address of the next sequential address (return address).
RPtarget- Target register pair RP???. The register holds a code address.
RA - Return Address Register- used in push and pops to determine if the return address (RA) should be pushed/popped on/from stack.
i - in this table B = byte; W = word
imm - Immediate value 8 bits for byte, 16 bits for word, 32 bits for double
immn - Immediate value of n bits:
Iposition- Bit number of data stored in memory
(8) - 8-bits e.g. the least significant byte of any R???
(16) - 16-bits e.g. a complete word register (R0..R11) or the least significant word of a double register (R12_L,R13_L,RA_L,SP_L)
abs - Absolute address - always unsigned
dispn - displacement of n bits - unsigned[a] for loads, stores and bitops, signed for branches

a. For CR16B compatibility LOAD/STOR (Rbase/RPbase) allow a negative displacement as well.

2.5 MEMORY ORGANIZATION

The CR16C provides access to 16M ofmemory. The memory is a uniform linear address space numbered sequentially starting at 0. CR16C 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.

Memory can be logically divided into the following regions (see Figure 2-4):

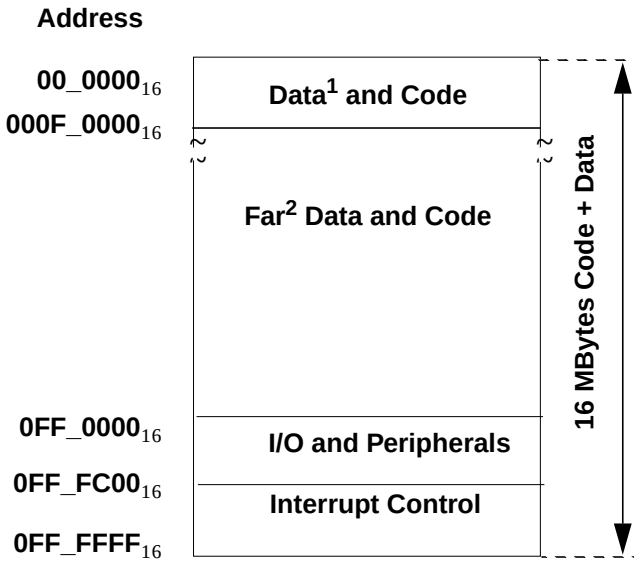
- 0 - 1M-64K (0 through $0E\_FFFF_{16}$)
  This region can be accessed efficiently for data-manipulation using all addressing modes. Therefore, it should be used for RAMs. There are no restrictions on code in this region.

- 1M-64K - 16M-64K ($0F\_0000_{16}$ through $0FE\_FFFF_{16}$)
  This region can be accessed for data-manipulation using all addressing modes. However there are some limitations and disadvantages when accessing data in this region (accessing data in absolute mode requires a longer instruction, a label in this region cannot be used as a displacement and moving or adding a data label to a register pair requires a longer instruction). There are no restrictions on code in this region. Therefore, use this address range for code and infrequently used data.

- 16M-64K - 16M-1K ($0FF\_0000_{16}$ through $0FF\_FBFF_{16}$)
  This region is reserved for I/O devices and peripherals. Loads, stores and bit operations using absolute addressing mode in this range are as efficient as in the range 0 - 1M-64K. This is because the core maps absolute addresses 0F_0000 through 0F_FFFF to 0FF_0000 through 0FF_FFFF.

- 16M-1K - 16M-1 ($0FF\_FC00_{16}$ through $0FF\_FFFF_{16}$)
  This region is reserved for the Interrupt Controller Unit and its acknowledge address, as well as other internal uses.

For more information on the addressing modes see Sections 2.6 and 6.2 as well as Appendix B.



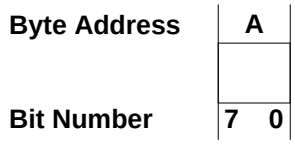
1 Efficient data access
2 Less efficient data access (Notation used in National CompactRISC Toolset. For notation used in other toolsets, see the appropriate documentation)

Figure 2-4. Memory Organization

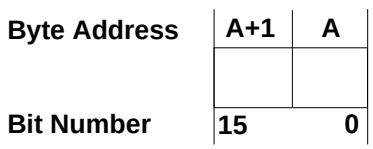
2.5.1 Data References

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

Figure 2-5. 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 2-5 shows the memory representation for data values. **Data Representation**



(a) Byte at Address A



(b) Word at Address A

in Memory

Data references

The CR16C supports references to memory by the **LOAD** and **STOR** instructions, as well as **TBIT**, **SBIT**, **CBIT**, **PUSH**, **POP**, **LOADM** and **STORM**. Bytes, words and double-words can be referenced on any boundary.

2.5.2 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*. The CR16C supports two types 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 exception handling. This information is automatically pushed, by the hardware, onto 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 for more information. The interrupt stack pointer is accessed via the ISP Register.

The program stack

The program stack is normally used by programs at runtime, 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. Note that this stack is handled by software only, e.g., the CompactRISC C Compiler generates code that pushes data onto, and pops data from, the program stack. Only **PUSH, POP** and **POPRET** instructions adjust the **SP** automatically; otherwise, software must manage the **SP** during save and restore operations.

The user and supervisor stacks

To support multi-tasking operating systems, support is provided for a two-level program stack: a user stack and a supervisor stack. The user stack pointer is accessed via the USP Register. The PSR contains a user mode status bit (PSR.U). When this user mode bit is set, the user stack pointer is used in place of SP for all operations. In addition, while in user mode, the PSR.U bit is read only. User mode is entered using the JUSR instruction, which causes a jump to the user program and entrance to user mode by setting the PSR.U bit. User mode is exited on any exception processing. While not in user mode, the user stack pointer can be accessed using the LPRD and SPRD instructions.

Using the user stack can save memory required for the stack, since supervisor information can be saved instead on the supervisor stack each time a task is saved.

An output signal is provided from the core (SFUSR) to indicate if the core is currently in user or supervisor mode. This may be used by a memory protection unit for checking accessed addresses in the various modes.

Both stacks expand toward address zero in memory.

2.6 ADDRESSING MODES

The CR16C supports these addressing modes: register/pair, immediate, relative, absolute, and indexaddressing. Memory is accessed by the generic load and store instructions, along with bit operations. These operations are supported with one or more of these modes.

When register pairs are used, the lower bits are in the lower index register and the upper bits are in the higher index register. When CFG.SR=0, the double-word registers R12, R13, RA and SP are also considered register pairs. For more details, see Section 2.3.1.

All references to register pairs should use parentheses. With a register pair, the lower numbered register pair should be on the right. For example,

```
jump (r5, r4)
load $4(r4,r3), (r6,r5)
load $5(r12), (r13)
```

Register/pair mode

In register/pair mode, the operand is located in a general-purpose register, or in a general-purpose register pair (see Section 2.3.1). For example, the following instruction adds the contents of the low order byte of register r1 to the contents of the low order byte of r2, and places the result in register r2.

```
ADDB r1, r2
```

Immediate mode

In immediate mode, the operand is a constant value which is specified within the instruction. For example, the following instruction multiplies the value of r4 by 4 and places the result in r4:

```
MULW $4, r4
```

Relative mode

In relative mode, the operand is obtained using a relative displacement value encoded in the instruction. This displacement is relative to the current Program Counter (PC) or general-purpose register or register pair.

In branch instructions, the displacement is always relative to the current value of the PC Register. For example the following instruction causes an unconditional branch to an address 10 ahead of the current PC.

```
BR *+10
```

In load, store and bit operations, the displacement value is relative to the contents of a general-purpose register or register pair. For example, the following instruction loads the data contained at an address 12 higher than the contents of r5 into r6.

```
LOADW 12(r5), r6
```

The following example loads the contents of memory at the address of the register pair (r5, r4) plus 4 into the register pair (r7, r6). r7 receives the high word and r6 receives the low word.

```
LOADD 4(r5, r4), (r7, r6)
```

Absolute mode

In absolute mode, the operand is located in memory, and its address is specified explicitly within the instruction (normally 20 or 24 bits). For example, the following instruction loads the byte at addresses 4000 into the lower 8 bits of register 6.

```
LOADB 4000, r6
```

Index mode

In index mode, the operand is located in memory. This mode supports load, store and bit operations, and is provided in order to handle relocatable code. CFG.SR must be 0 to use this addressing mode. Register R12 or R13 is used to hold a base index address to which absolute or relative mode addresses are added. For example the following instruction loads the word at address (r12) + (r5,r4) + 4 into r6.

```
LOADW [r12]4(r5,r4), r6
```

- For relative mode instructions, the memory address is obtained using the value of either R12 or R13, and adding the value of a register pair and a displacement. The displacement can be a 14 or 20-bit unsigned value, which is encoded in the instruction.
- For absolute mode instructions, the memory address is obtained using the value of either R12 or R13, and adding a 20-bit absolute address label which is encoded in the instruction.

For a more detailed explanation, see Section B.2 on page B-2.

For data addressing, Table 2-1 summarizes the size of the instruction along with the displacement sizes supported. Note that index addressing mode is not available when CFG.SR is set.

Table 2-1. Instruction Sizes Supported

Instructions	Data relative to	Instruction Size		
		1 Word	2 Words	3 Words
Load/Store (b/w/d)	register pair	disp4	disp16	disp20
	register	disp0[a]	disp14[a]	disp20
	index[b]	disp0	disp14, abs20	disp20
	abs		abs20	abs24

Instructions	Data relative to	Instruction Size		
		1 Word	2 Words	3 Words
(c/s/t)bit(b/w) Store Imm	register pair	disp0	disp16	disp20
	register		disp14[a]	disp20
	index[b]		disp14, abs20	disp20
	abs		abs20	abs24

a. when CFG.SR = 1
b. when CFG.SR = 0

The data addressing modes supported for all instructions are identical in two and three-word formats. The instructions encoded in one word give priority to addressing relative to a register pair.

This page is intentionally blank.

Chapter 3
EXCEPTIONS

This chapter briefly describes exceptions and how they are handled, and provides detailed information on exception processing.

3.1 INTRODUCTION

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 and handling

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 CR16C recognizes twelve exceptions: nine 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 multi-programming and multi-tasking (including real-time) capabilities.

Exception-handling in the CR16C 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 and extract the handler address.

Stack types

The CR16C features an interrupt stack, a supervisor stack and a user stack. The processor uses the interrupt stack solely for saving the PC and the PSR Register values during exception processing. This process occurs in hardware, without software intervention. The software uses the supervisor and user stacks to save register values and to pass parameters upon subroutine entry and subroutine calls. These stacks are managed by software using the **PUSH, POP and POPRET** instructions.

This stack architecture provides the following benefits:

- The essentials of the processor's state (PC and PSR) are saved correctly on the interrupt stack, even during nested, non-maskable interrupts. This process does not need to rely on disabling interrupts to allow software to save PC and PSR values on the interrupt stack.
- As the processor saves just the PC and PSR when exceptions occur, interrupt latency is kept at a minimum. During exception handling, the software need only save the registers it modifies, thus minimizing interrupt response time, and saving memory.

The exception process

When an exception occurs, the CPU automatically preserves the state of the program immediately prior to the handling of the exception: a copy of the PC and the PSR is made and pushed onto the interrupt stack. The contents of the PSR is adjusted for exception processing. 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 "RETX Return from Exception" on page 5-52.

3.2 INTERRUPT HANDLING

The CR16C provides three types of interrupts: non-maskable (NMI), maskable, and In-System Emulator (ISE).

Non-Maskable Interrupts (NMI)

NMI are used for events which require immediate handling to preserve system integrity (such as an imminent power failure), and cannot be disabled. NMI use vector number 1 in the dispatch table. When an NMI is detected, the CR16C performs an interrupt-acknowledge bus cycle to address $0FF\_FF00_{16}$, and discards the byte that is read during that bus cycle.

Maskable interrupts

Maskable interrupts are disabled whenever PSR.E or PSR.I are 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 Enable Maskable Interrupts" on page 5-20 and "DI Disable Maskable Interrupts" on page 5-19). PSR.E is 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 vector number by performing an interrupt acknowledge bus cycle in which a byte is read from address $0FF\_FE00_{16}$. This byte contains a number in the range 16-127 (the vector), which is used as an index in the dispatch table to find the address of the appropriate interrupt handler. Control is then transferred to that interrupt handler.

In-System Emulator (ISE) interrupts

ISE interrupts cannot be disabled; they are reserved for system debug implementation. ISE interrupts use vector number 15 in the dispatch table. When an ISE interrupt is detected, the CR16C performs an interrupt-acknowledge bus cycle to address $0FF\_FC00_{16}$, and discards the byte that is read during that bus cycle.

3.3 TRAPS

The CR16C recognizes the following traps:

Breakpoint (BPT)

BPT is used for program debugging. Caused by the **EXCP BPT** instruction.

Supervisor Call (SVC)

SVC temporarily transfers control to supervisor software, typically to access facilities provided by the operating system. It is caused by the **EXCP SVC** instruction.

Flag (FLG)

FLG Indicates various computational exceptional conditions. it is caused by the **EXCP FLG** instruction.

Division by Zero (DVZ)

DVZ indicates an integer division by zero. It is caused by the **EXCP DVZ** instruction, which can be used by integer division emulation code to indicate this exception.

Undefined Instruction (UND)

UND indicates undefined opcodes. It is 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).
- an LPR/LPRD when PSR.U bit is set.

Illegal Address (IAD)

IAD indicates that an illegal address was detected. For the CR16C, this trap is generated whenever an address outside the address range of 0 to 16M-1 is detected.

Wraparound can happen either when the 16M boundary is crossed in a positive direction or when the 00__0000 address is crossed in a negative direction - and the resulting address yields in the illegal space. Both are trapped and flagged as illegal operations.

This is done primarily for reasons of future compatilbity so that the address range of the CR16C can easily be increased beyond 16M to maintain binary compatibility, without any issues of wraparound compatibility.

The CR16C also provides an optional input for external illegal address detection, which can be used in the system to protect (unused) memory areas from being accessed by the core - this is not required in an AHB system like the CR16CPlus which provides an error response on the bus.

Trace (TRC) TRC occurs before an instruction is executed when the PSR.P bit is 1. It is used for program debugging and tracing. See Chapter 4.

Debug (DBG) 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. It is used for instruction-execution and data-access breakpoints. See Chapter 4.

BPT, TRC and DBG DBG, TRC and BPT traps can also generate an interrupt acknowledge cycle for observability purposes, to alleviate the design of an ISE. 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 $0FF\_FC02_{16}$ (DBG), $0FF\_FC0C_{16}$ (TRC) and $0FF\_FC0E_{16}$ (BPT) respectively. See Chapter 4.

3.4 DETAILED EXCEPTION PROCESSING

3.4.1 Instruction Endings

The CR16C 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 DVZ trap, are acknowledged during execution of an instruction. In such a case, the instruction is suspended. See Table 3-2.

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 saved on the interrupt stack contains the address of the suspended instruction.

3.4.2 The Dispatch Table

The CR16C recognizes eight 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 that contains 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 CR16C supports a dispatch table with either 16-bit or 32-bit entries. The 16-bit version is used when memory is at a premium and exception handlers can be restricted to the first 128 K of memory. Otherwise, the 24-bit entry dispatch table is used, thus removing all restrictions on the location of the exception handlers. In 16-bit mode, each entry occupies one word of memory, containing bits 1 through 16 of the exception handler's start address; in 24-bit mode, each entry occupies two adjacent words in memory, the first holding bits 1 through 16, and the second holding bits 17 through 23 of the exception handler address, right justified. This mode selection is determined by the CFG.ED configuration bit:

- (CFG.ED = 0) selects the 16-bit entry dispatch table.
- (CFG.ED = 1) selects the 32-bit entry dispatch table. Entries in the dispatch table are 32 bits wide but only the lower 24 bits are used.

There are no restrictions on the location of the dispatch table in memory. This is determined by INTBASEH and INTBASEL, which are accessed as processor registers.

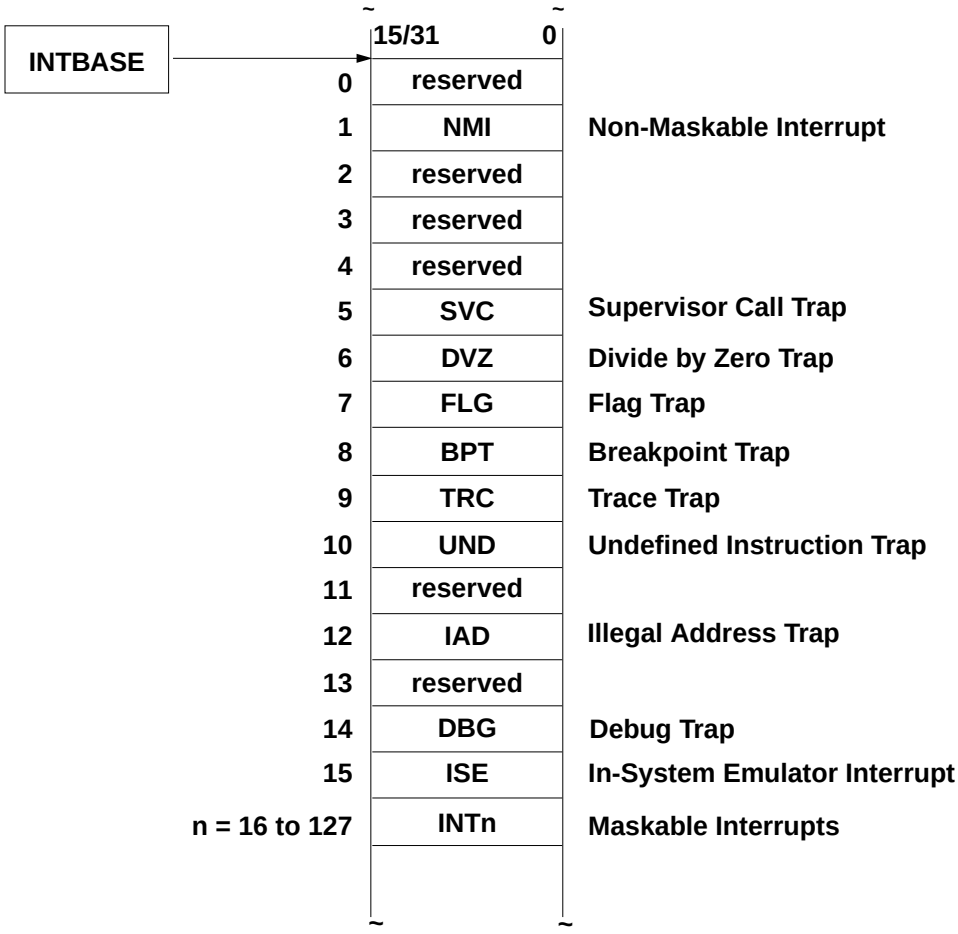


Figure 3-1. Dispatch and Jump Table

3.4.3 Acknowledging an Exception

The CR16C performs the following operations in response to interrupt and trap exceptions:

1. Decrements the Interrupt Stack Pointer (ISP) Register by 6.

2. Saves the contents of the current PSR on the highest-address word of the interrupt stack; saves the current value of the PC on the next two words of the interrupt stack. See Figure 3-2.

3. Alters PSR by clearing the control bits, as shown in Table 3-2.

4. For interrupts, displays information during the interrupt acknowledge bus cycle to indicate the type of interrupt encountered via a byte read from the address shown in Table 3-1.

Table 3-1. Interrupt ACK Vector Address

Address	Type	Description
FF_FE00	INT	maskable interrupt - mapped to ICU
FF_FF00	NMI	non-maskable interrupt; for observability
FF_FC00	ISE	in-system emulation trap
FF_FC02	ADBG	alternate debug trap
FF_FC0C	ATRC	alternate trace trap
FF_FC0E	ABPT	alternate break point trap

For NMIs or traps, the exception number is set as the vector, to be used while accessing the dispatch table.

5. When CFG.ED is 1, reads the dispatch table double-word entry at address (INTBASE) + vector × 4. When CFG.ED is 0, reads the single-word entry from (INTBASE) + vector × 2

   The dispatch table entry is used to call the exception handler and is interpreted as a pointer that is loaded into bits 1 through 16. If CFG.ED = 1, it is also loaded into bits 17 through 23 of the PC.

   Bit 0 of the PC is cleared. See Figure 3-3.
   The pointer is stored in the dispatch table in little-endian format. For a double-word entry, the lower address word contains address bits 1 through 16, and the upper address word contains address bits 17 through 23.

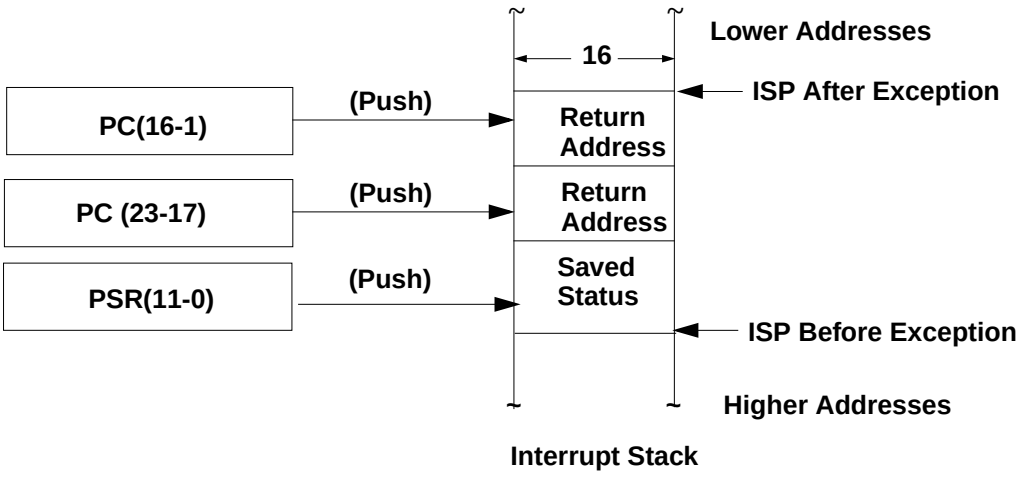


Figure 3-2. Saving PC and PSR Contents During Exception Acknowledge

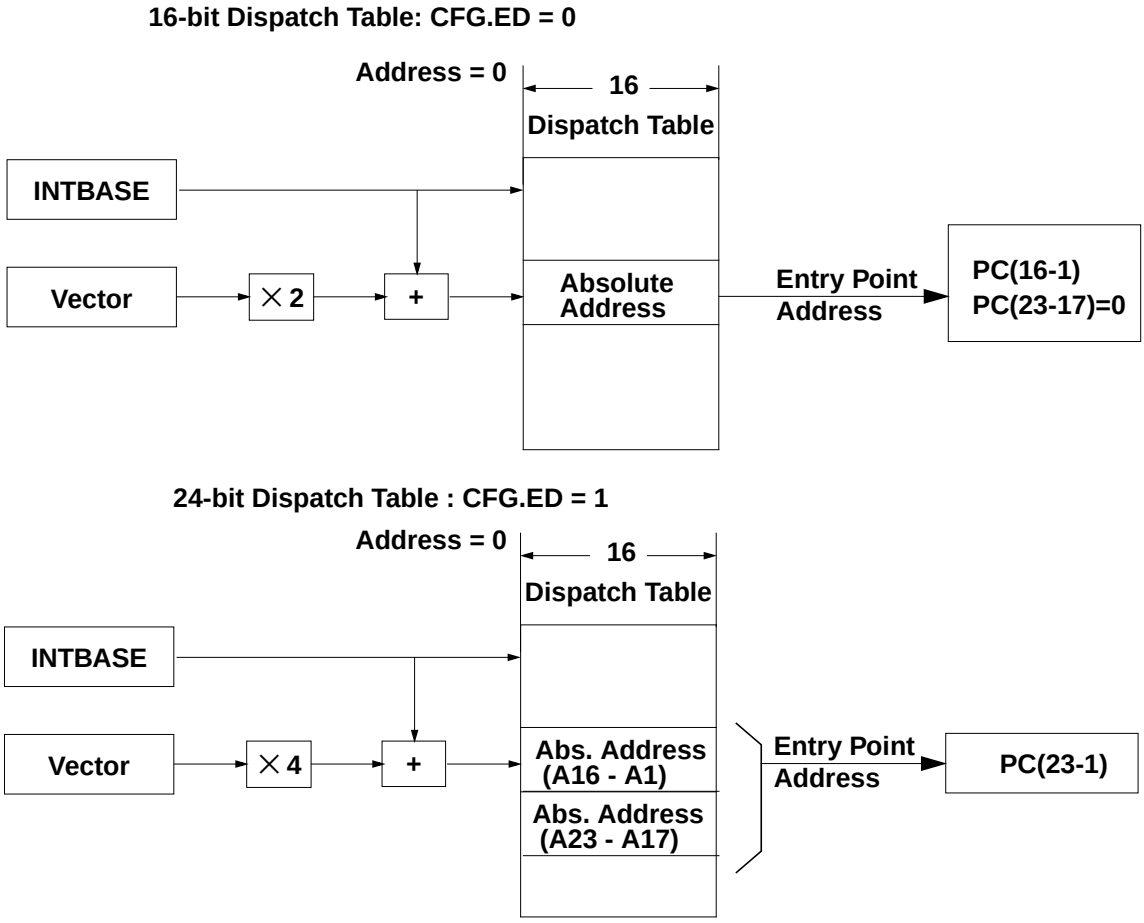


Figure 3-3. Transfer of Control During an Exception Acknowledge Sequence

Table 3-2 summarizes how each type of exception is acknowledged.

Table 3-2. 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 U
BPT, **DVZ**, **FLG**, **SVC**	Suspended	Current	None[a]	P T U
UND	Suspended	Current	None[a]	P T U
TRC	Before start of instruction	Next	None, P	P T U
IAD	Before start of instruction	Next	None	P T U
DBG	Before start of instruction	Next	None	I P T U

a. If a TRC trap is used in conjunction with these exceptions, the trap handler may need to clear the P bit of the PSR, which is saved on stack, to prevent a redundant trace exception. For more information, refer to Sections 3.4.5 and 4.1.1 on page 4-1.

3.4.4 Exception Service Procedures

After the CR16C 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 IAD, 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, the exception service procedure must be programmed either to simulate a suspended instruction or to retry execution of a 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. The CR16C begins executing the instruction following the suspended instruction. For example, when a UND trap 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 CR16C 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.

3.4.5 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 5-52.

3.4.6 Priority Among Exceptions

The CR16C checks for specific exceptions at various points while executing an instruction (see Figure 3-4).

If several exceptions occur simultaneously, the CR16C responds to the exception with the highest priority.

If several maskable interrupts occur simultaneously, the Interrupt Control Unit (ICU) determines the highest priority interrupt, and requests the CR16C to service this interrupt.

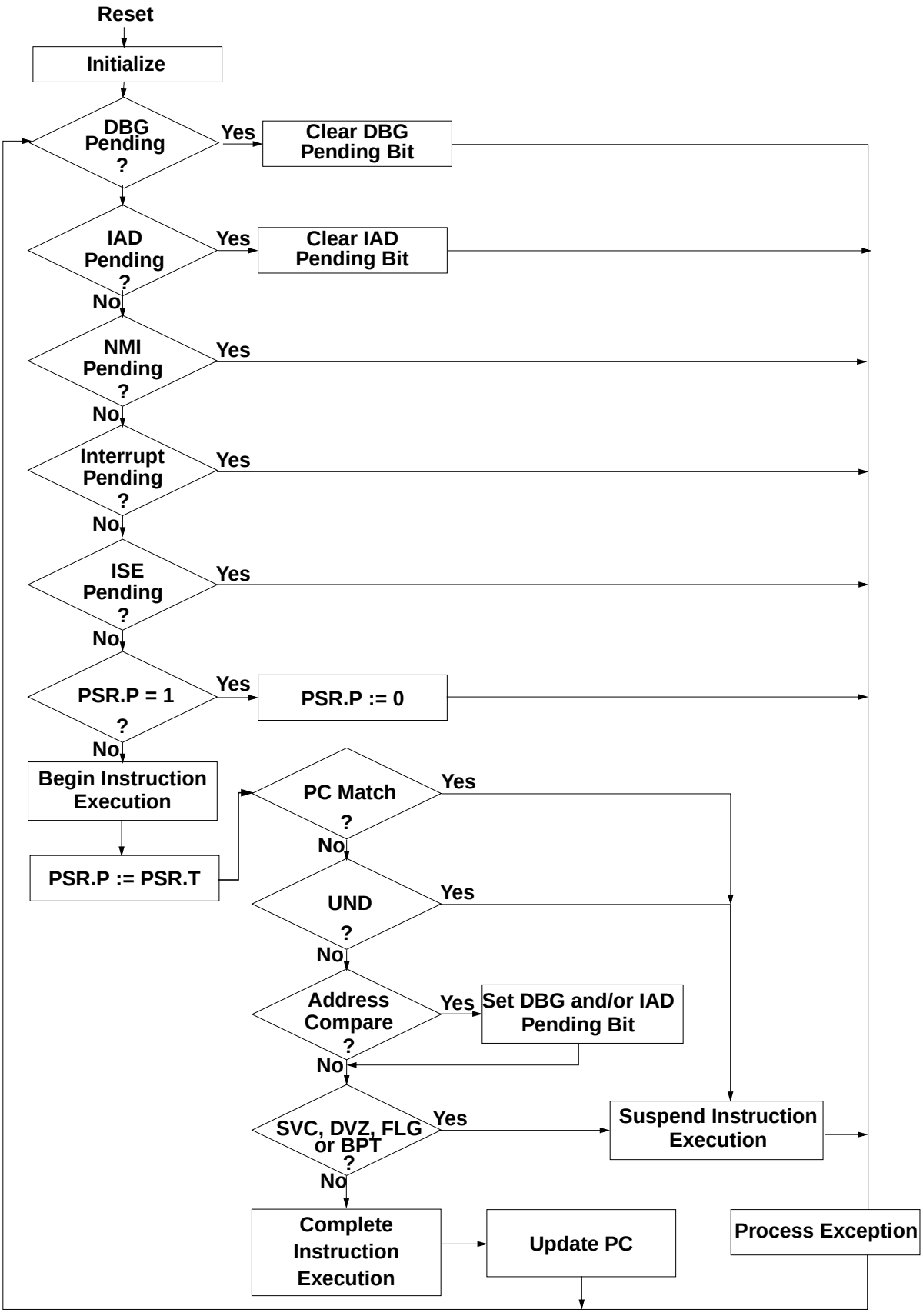


Figure 3-4. Exception Processing Flowchart

Before executing an instruction

Before executing an instruction, the CR16C checks for pending DBG, IAD, interrupts and TRC 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 TRC trap is pending), then the CR16C clears PSR.P and processes the TRC trap.

If no TRC trap or interrupt is pending, the CR16C begins executing the instruction by copying PSR.T to PSR.P. While executing an instruction, the CR16C may detect a trap.

During executing an instruction

First, the CR16C checks for a UND trap or a PCMATCH (if enabled), then it looks for any of the following mutually exclusive traps: SVC, DVZ, FLG or BPT. The CR16C 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 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 and address error conditions. If the CPU detects an enabled address-compare, a DBG trap remains pending until after the instruction is completed. If an invalid address is detected, the IAD trap becomes pending and the CPU responds to it before executing the next instruction.

3.4.7 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 3-2), 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. 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 NMI and nested ISE interrupts are always serviced.

Single-stepping through instructions (TRC) is also subject to interrupt prioritizing, as the P and T bits are both cleared when exceptions are being handled. Therefore, exceptions occurring during an instruction are served before the single-step handler (TRC).

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

3.5 RESET

A reset occurs when the appropriate signal is activated. Reset must be used at power-up to initialize the CR16C.

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 have not yet ended, may not be written to their destinations.
- Any pending interrupts and traps are eliminated.
  For correct operation, maskable interrupts as well as ISE and NMI, must be inactive by the time the reset signal is deactivated, to avoid being serviced immediately after the reset period.

Upon reset, the following operations are executed:

- PC(16-1) current values are stored in R0, and PC(23-17) current values are stored in R1.
- The current value of the PSR is stored in R2.
- The following internal registers are cleared to 0: PC, CFG, and PSR, except for PSR.E, which is set to 1.
- When no hardware debug module is present in the system, processor registers DBS and DCR are also cleared to 0. Otherwise, refer to the hardware debug documentation for information on clearing these registers.

After reset, the processor begins normal execution at memory location 0 (or another memory location when set by hardware). The reserved bits in these registers and the contents of all other registers are unpredictable.

Note that some uncertainty exists as to which PC is saved at reset in R0, R1 and R2. In case of sequential instructions, the address saved may also be that of the next sequential instruction to be executed, as well as the currently interrupted instruction. In case of instruction flow change (e.g. **BR**, **BAL**, **JAL**), the address saved may also be that of the jump or branch target.

This page is intentionally blank.

Chapter 4
ADDITIONAL TOPICS

This chapter discusses debugging support, the instruction execution order and cache support for the CR16C.

4.1 DEBUGGING SUPPORT

The following CR16C features make program debugging easier:

- Instruction tracing
- Soft break generation by breakpoint instruction (**EXCP BPT**)
- User programmable breakpoint features:
  - Instruction address match
  - Instruction address range match
  - Data access
  - Data access range
  - Combination of the above (complex breakpoints)
- External instruction tag during fetch to break before execution
- ISE support signals

The Processor Status Register (PSR), the Debug Base Register (DBS), the Debug Control Register (DCR), the Debug Status Register (DSR), and the Compare-Address Registers (CAR) control/s and monitor the debug and trace features.

4.1.1 Instruction Tracing

The following paragraphs describe the instruction tracing support via software for a resident monitor program. These features are only available if no external hardware debug module is present. If it is, tracing is controlled solely by this module; the respective PSR bits have no effect.

Instruction tracing can be used during debugging to single-step through selected portions of a program. The CR16C uses two bits in the PSR to enable and generate TRC traps. Tracing is enabled by setting the T bit in the PSRl.

During the execution of each instruction, the CR16C copies the PSR.T bit into the PSR.P (trace pending) bit. Before beginning the next instruction, the CR16C checks if the PSR.P bit is 1 to determine whether a TRC trap is pending. If yes, the CR16C generates a TRC trap before executing the instruction.

For more information on the different exception priorities, see Figure 3-4. on page 3-11.

If any other trap or interrupt is requested during execution of a traced instruction, its entire service procedure is allowed to complete before the TRC trap occurs.

For example, if a 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 saved 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 performs 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.

4.1.2 The Breakpoint Instruction

Debuggers can use the breakpoint instruction (**EXCP BPT**) to stop the execution of a program at specified instructions in order to examine its status. The debugger replaces these instructions with the breakpoint instruction. It then starts 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.

When an external hardware debug module is present, the CR16C enters the hardware debug mode upon execution of the **EXCP BPT**.

4.1.3 User Programmable Breakpoint Features

The CR16C provides a set of debug registers which allows the user to enable various debug features. These features are controlled by the following five core registers: DBS, DCR, DSR, CARn and CARn+1. Loading and storing these registers is accomplished using the **LPR/LPRD** and **SPR/SPRD** instructions.

The basic debug register set (DCRb, DSRb, CARn and CARn+1, where b refers to the debug register bank and n refers to the debug channel) allows the control of two debug channels. Depending on the configuration, additional debug registers can be mapped in the same space to control an additional two channels each.

When the optional hardware debug module is present, the debug control and status registers are no longer accessible by the core. Instead, they are solely controlled by the hardware debug module.

The following functions are available via the debug registers:

- Single Instruction Breakpoint
  Breakpoint/Watchpoint on execution of an instruction, fetched from one specific address. Requires a single debug channel.
- Masked Instruction Breakpoint
  Breakpoint/Watchpoint on execution of an instruction, fetched from a small address range. The address range is defined by one Compare Address Register (CARn) in conjunction with two mask bits to mask out up to three of the LSBs of the CARn register. This breakpoint requires a single debug channel.
- Instruction Breakpoint Range
  Breakpoint/Watchpoint on execution of an instruction, fetched from a specific address range. The address range is defined by utilizing an address compare register pair to specify a start and an end address. Requires two debug channels.
- Single Data Access Breakpoint
  Breakpoint/Watchpoint on a data access (read/write/read-or-write) to one specific address. Requires a single debug channel.
- Masked Data Access Breakpoint
  Breakpoint/Watchpoint on a data access (read/write/read-or-write) to a small address range (up to 8 bytes). The address range is defined

by one Compare Address Register (CARn) in conjunction with two mask bits to mask out up to three of the least significant bits of the CARn register. Requires a single debug channel.

- Data Access Breakpoint Range
  Breakpoint/Watchpoint on a data access (read/write/read-or-write) to a specific address range. The address range is defined by utilizing an Compare Address register pair to specify a start and end address. Requires two debug channels.
- Complex Breakpoint
  Breakpoint/Watchpoint upon the detection of a complex breakpoint/watchpoint condition. Two CARs are combined to specify the conditions of a complex breakpoint/watchpoint.

Instruction Breakpoints (PC Match Breakpoints)

The CR16C provides instruction breakpoints by comparing an address present on the address bus, during an instruction fetch bus cycle, with a compare address values stored in the internal CARs (CARn, n=0...7).

If the CR16C fetches an instruction from an address and an instruction breakpoint has been set, this instruction is internally "tagged" along with the fetch. This tag follows the instruction though the pipeline. When this instruction reaches the execution stage of the pipeline, the CR16C enters debug mode before the CR16C starts executing this instruction.

Single instruction breakpoint

The breakpoint logic can be set up to generate a breakpoint upon the execution of an instruction, fetched from one specified address. This address, used for comparison, can be specified by one Compare Address Register (CARn). In addition, the configuration bits in the corresponding Debug Control Register (DCRn) must be set up to enable a single instruction breakpoint as follows:

- The PC Match enable bit (PCn) must be set to 1 to select an instruction breakpoint.
- The Address Mask bits (MSKn) must be cleared to 0.
- The Enable Address Range Breakpoint (EARn) bit must be cleared to 0 to disable an address range breakpoint.
- The Data Write and Data Read Address Match Enable bits (DWRn and DRDn) must be cleared to 0 to disable a data access breakpoint.
- The Debug Enable bit (DEN) must be set to 1 to enable the breakpoint logic.

Figure 4-1 shows an example for using the compare logic x and y for the detection of two single instruction breakpoints.

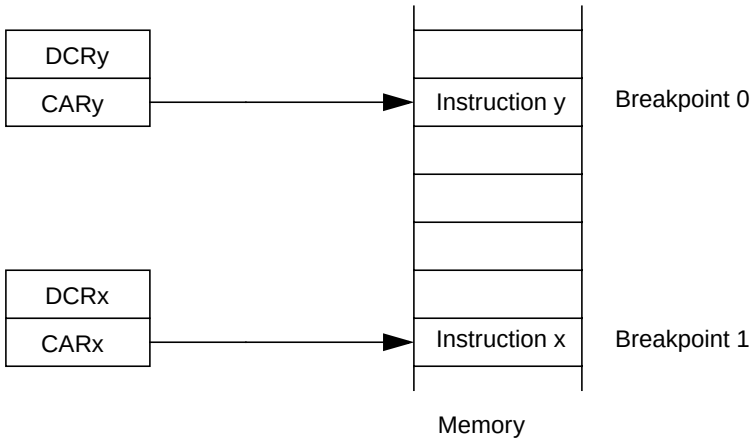


Figure 4-1. Single Instruction Breakpoint

Masked instruction breakpoint

The breakpoint logic can be set up to generate a breakpoint upon the execution of an instruction, fetched from a small address range, specified by the address value stored in CARn in conjunction with two Address Mask bits (MSKn). These bits mask out up to three of the least significant bits of CARn. The masked-out CARn bits are then treated as "don't care" by the address compare logic. This allows the tool to define a break range of up to 8 bytes by only utilizing one CAR. The DCRn must be configured as follows to enable a masked instruction breakpoint:

- The PCn bit must be set to 1 to select an instruction breakpoint.
- The MSKn[1:0] bits must be set to select an address range of the desired size.
- The EARn bit must be cleared to 0 to disable an address range breakpoint.
- The DWRn and DRDn bits must be cleared to 0 to disable a data access breakpoint.
- The DEN bit must be set to 1 to enable the breakpoint logic.

Note: The least significant bit of the address bus during instruction fetches is always 0, as instruction fetches are always performed on word-aligned addresses.

Figure 4-2 shows an example for using the compare logic n for the detection of a masked instruction breakpoint, covering two one-word instructions, occupying 4 bytes of address locations in memory.

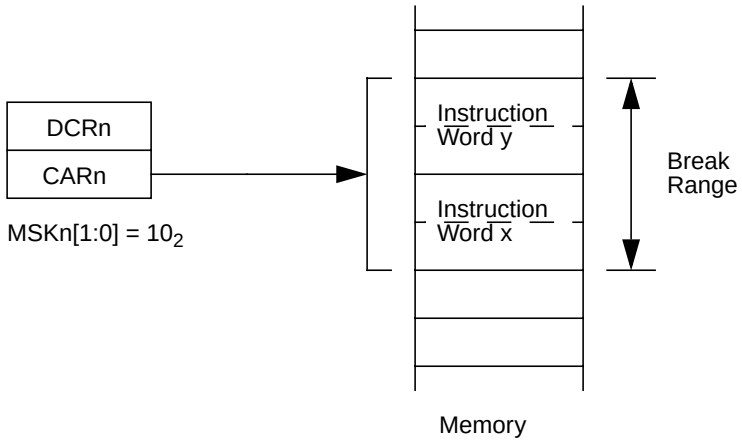


Figure 4-2. Masked Instruction Breakpoint (4-Byte Address Range = Two Instruction Word Range)

Instruction breakpoint range

The breakpoint logic can be set up to generate a breakpoint upon the execution of an instruction fetched from a specified address range. In order to specify this address range, two CARs (CARn and CARn+1, with n being an even number) are combined. The configuration bits in (DCRn+1) are ignored. Only the configuration bits in DCRn must be set up to enable the conditions for an instruction breakpoint range, as follows:

- The PCn bit must be set to 1 to select an instruction breakpoint.
- The MSKn[1:0] bits must be cleared to 0.
- The EARn bit must be set to 1 to enable an address range breakpoint.
- The DWRn and DRDn bits must be cleared to 0 to disable a data access breakpoint.
- The DEN bit must be set to 1 to enable the breakpoint logic.

Figure 4-3 shows an example for using compare logic n and m for the detection of a range instruction breakpoint between (and including) instructions x and y.

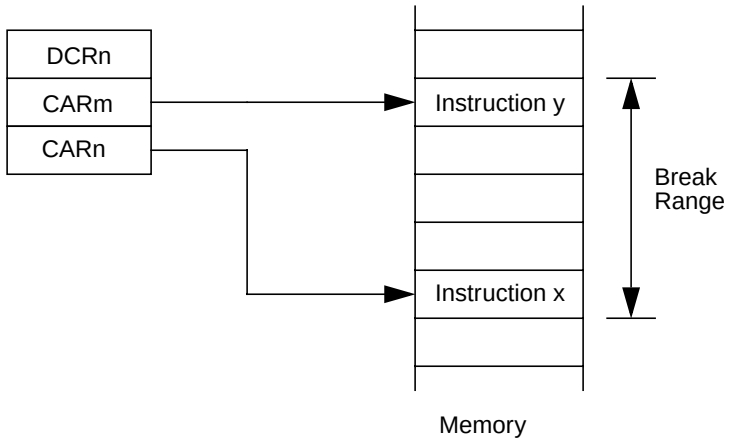


Figure 4-3. Instruction Breakpoint Range

Instruction Watchpoints

Note: This feature can only be used with the optional external hardware debugger SDI+.

The address compare logic can be used for detecting watchpoints instead of breakpoints. The instruction watchpoint works the same as the instruction breakpoint with this exception: when the “tagged” instruction reaches the execution stage of the pipeline, program execution continues in real-time, instead of stopping, while the CR16C asserts the respective SFDBGEVP output.

In addition, the Breakpoint/Watchpoint pending flag in the DSR is set to 1 to indicate a watchpoint hit. The watchpoint hit is flagged by the SFDBGEVP output and the Watchpoint pending bit, as soon as the instruction has been completed.

To specify an instruction watchpoint, the Watchpoint Enable bit (WPEn) must be set to 1, along with the other configuration bits used to specify watchpoint conditions (see “WPEn” on page 4-18).

Data Access Breakpoints

The CR16C is capable of detecting data access breakpoints by comparing an address present on the address bus during a data transfer bus cycle with a compare address value stored in one of the internal CARs (CARn, n=0…7).

If the CR16C performs a data transfer to/from an address for which a data access breakpoint has been set, the CR16C enters debug mode after it has completed execution of this instruction (i.e., after all data for this instruction has been transferred).

Optionally, the respective SFDBGEVP output can also be utilized to signal this event to an external hardware debug tool.

The breakpoint logic can be set up to trigger a breakpoint either on a data read access, a data write access, or both.

Single data access breakpoint

The breakpoint logic can be set up to generate a breakpoint upon execution of a data transfer from/to one specified address. This address, used for comparison, can be specified by one CARn. In addition, the configuration bits in the corresponding DCRn must be set up to enable the desired data access breakpoint, as follows:

- The DWRn and/or DRDn bits must be set to 1 to disable a data access breakpoint on a data read access, a data write access or both.
- The PCn bit must be cleared to 0 to disable an instruction breakpoint.
- The MSKn[1:0] bits must be cleared to 0.
- The EARn bit must be cleared to 0 to disable an address range breakpoint.
- The DEN bit must be set to 1 to enable the breakpoint logic.

Figure 4-4 shows an example for using the compare logic n and m for the detection of two, independent data access breakpoints at the address of breakpoint x and y respectively.

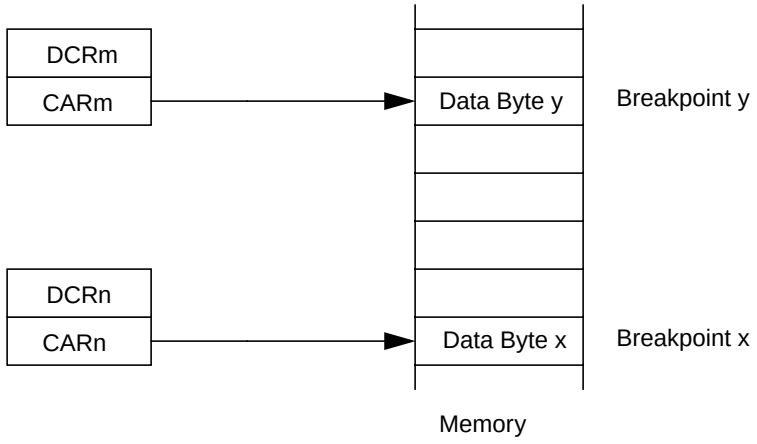


Figure 4-4. Single Data Access Breakpoints

Data transfers include all transfers but instruction fetches.

Masked data access breakpoint

The breakpoint logic can be set up to generate a breakpoint on the data transfer from or to a small address range, specified by the address value stored in the CARn in conjunction with two address mask bits (MSKn). These mask bits mask out up to three least significant bits of CARn. The masked-out CARn bits are then treated as “don’t care” by the address compare logic. This allows definition of a breakpoint range of up to 8 bytes by only utilizing one CAR. This can be useful to set up a data access breakpoint to a data structure which occupies up to 8 bytes in memory. DCRn must be configured as follows to enable a masked data access breakpoint:

- The DWRn and DRDn bits must be set to enable the desired type of data access breakpoint.
- The MSKn[1:0] bits must be set to select an address range of the desired size.
- The EARn bit must be cleared to 0 to disable an address range breakpoint.
- The PCn bit must be cleared to 0 to disable an instruction breakpoint.
- The DEN bit must be set to 1 to enable the breakpoint logic.

Figure 4-5 shows an example for using the compare logic n for the detection of a masked instruction breakpoint, covering 4 bytes of data in memory.

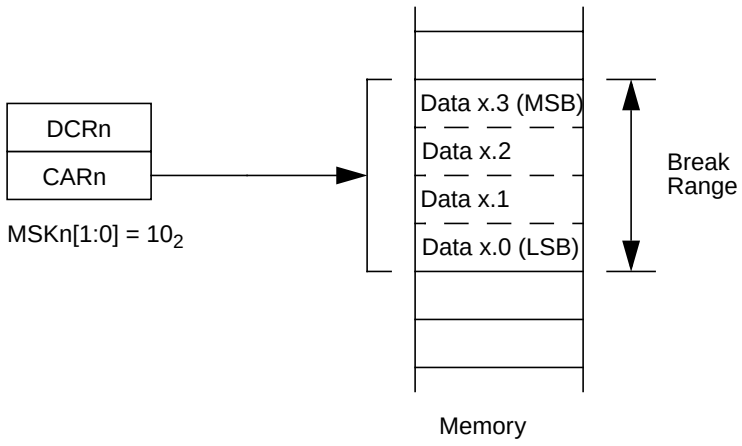


Figure 4-5. Masked Data Access Breakpoint (e.g for Double Data Type)

Data access breakpoint range

The breakpoint logic can be set up to generate a breakpoint upon the execution of a data transfer from/to one specified address range. In order to specify this address range, two CARs (CARn and CARn+1 with n being an even number) are combined. The configuration bits in the (DCRn+1) are ignored. Only the configuration bits in the DCRn must be set up to enable the conditions for the data access breakpoint range, as follows:

- The DWRn and/or DRDn bits must be set to 1 to disable a data access breakpoint on a data read access, a data write access or both.

- The EARn bit must be cleared to 0 to disable an address range breakpoint.
- The PPCn bit must be cleared to 0 to disable an instruction breakpoint.
- All three MSKn bits must be cleared to 0.
- The DEN bit must be set to 1 to enable the breakpoint logic.

Figure 4-6 shows an example for using the compare logic n and m for the detection of an access to data within a range between (and including) address x and y.

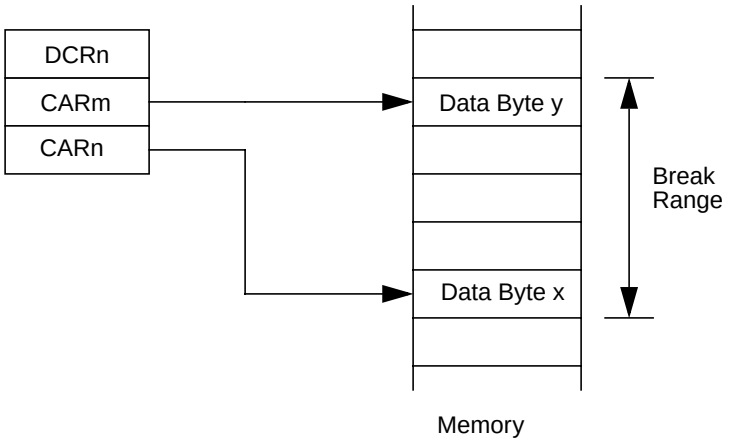


Figure 4-6. Data Access Breakpoint Range

Data Access Watchpoints

Note: This feature can only be used with the optional external hardware debugger SDI+.

The address compare logic can be used for detecting watchpoints instead of breakpoints. The data access watchpoint works the same as the data access breakpoint with this exception: when the specified location is accessed, program execution continues in real-time, instead of stopping, while the CR16C asserts the respective SFDBGEVP output.

In addition, the Breakpoint/Watchpoint pending flag in the Debug Base register (DBS) is set to 1 to indicate a watchpoint hit. The watchpoint hit is flagged by the SFDBGEVP output and/or the Watchpoint pending bit as soon as the instruction, which performs the data access(es), has been completed.

To specify a data access watchpoint, the Watchpoint Enable bit (WPn) must be set to 1, along with the other configuration bits used to specify the watchpoint conditions (see “Data Access Watchpoints” on page 4-10).

Complex Breakpoints

Two CARs can also be combined to form a complex breakpoint. A complex breakpoint hit is a sequence of two successive breakpoints. When the first breakpoint condition is confirmed as true, this event is held pending while the CR16C continues operation without interference. When, subsequently, the second breakpoint condition occurs, the CR16C is stopped and enters debug mode if an external debug module is present. When no external debug module is present, a DBG trap is triggered.

In order to define both breakpoint conditions, the same pair of CARs is used as for breakpoint ranges. However, both DCRs are used to define the two breakpoint conditions. Thus, a complex breakpoint can be defined as:

- A sequence of two instruction breakpoints (single and masked)
- A sequence of two data access breakpoints (single and masked)
- A sequence of an instruction breakpoint followed by a data access breakpoint (single and masked).
  This type of breakpoint can be used, for example, to break on a data access, performed by a specified task (instruction).
- A sequence of a data access breakpoint followed by an instruction breakpoint (single and masked).

To specify a complex breakpoint, the address compare logic for both breakpoints should be set up as follows:.

- The PCn, DWRn and/or DRDn bits must be set to enable the desired type of breakpoints.
- The WPn bit must be cleared to 0
- The MSKn[1:0] bits must be set to specify the desired range.
- The Complex Breakpoint Enable bit (ESQ) must be set to 1.
- The EARn bit must be cleared to 0.
  (Note: When the ESQ bit is set to 1, it overrides the EAR bit if also set to 1. This enables a complex breakpoint and disables a breakpoint range.
- The DEN bit must be set to 1 to enable the watchpoint logic.

Figure 4-7 shows the sequence logic for complex breakpoints.

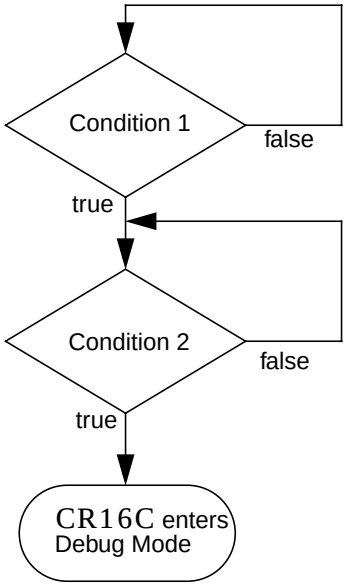


Figure 4-7. Complex Breakpoint Detection Flow

External Tag on Fetch

The CR16C 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, the 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 DBG exception takes precedence over other exceptions. Since DBG execution masks all the maskable exceptions, the trace exception (TRC) and any regular pending interrupt (INTP) which occurred during the previous instruction, are delayed until after the DBG handler has ended.

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.

When an external hardware debug module is present, the CR16C enters the hardware debug mode before the tagged instruction is executed.

4.1.4 Example Breakpoints

To improve external hardware debug support, the CR16C on-chip debug capabilities are slightly different from previous CR16 cores.

The following summarizes the possible breakpoint options and shows the recommend software actions:

1. Exit the DBG-handler and continue at the same address
   (retain Breakpoint configuration)
   -> set DCR.DEN = 1 (DCR = 0x90)

2. Exit the DBG-handler and continue at another address
   (retain Breakpoint configuration)
   -> clear DCR.PC
   -> set DCR.PC = 1 and DCR.DEN = 1 (DCR = 0x90)

3. Exit the DBG-handler and disable Breakpoint
   -> clear DCR.PC

4. Exit the DBG-Handler and enable TRACE at the same address
   (retain Breakpoint configuration)
   -> set DCR.DEN = 1 (DCR = 0x90)

5. Exit the DBG-Handler and enable TRACE at another address
   (retain Breakpoint configuration)
   -> clear DCR.PC
   -> set DCR.DEN = 1 (DCR = 0x90)

6. Exit the DBG-Handler and enable TRACE and disable Breakpoint
   -> clear DCR.PC

Note the difference between examples 1 and 2, and between 4 and 5. This is a result of the break-before-execution logic that makes it necessary for the CR16C to remember if a breakpoint has occurred so that the instruction can be executed on resuming operation. A register, called the breakpoint mask, is used for this purpose. If execution continues without a RETX, i.e., the instruction does not get executed, the breakpoint mask remains set and prevents a breakpoint on the next occurrence.

To clear the breakpoint mask, reset the PC bit. To keep the breakpoint, set the DEN bit. To clear the breakpoint, clear both the PC and DEN bits.

The CR16C core works with the following sequence:

Execute until a hardware break occurs:

- Option 1 (same address):
  - Set DCR.PC = 1; DCR.DEN = 1 (i.e., set DEN again)
  - Go (RETX - *same address* - the instruction where the Breakpoint occurred is executed)
- Option 2 (different address):
  - Clear DCR.PC = 0; DCR.DEN = don't care (clear breakpoint mask)
  - Set DCR.PC = 1; DCR.DEN = 1 (i.e., set PC/DEN again)
  - Modify PC on stack
  - Go (RETX - *different address*)

4.1.5 In-System Emulator (ISE)

The CR16C core supports the development of real-time ISE equipment and Application Development Boards (ADBs) with the following features:

- 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
- A BRKL line for accurate external breakpoints on instruction execution, by tagging instructions during the fetch cycle
- Forcing INTBASE to 0 for the ISE, DBG, TRC and BPT exceptions, programmable by the DCR
- A set-only Lock bit in DCR to protect 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 a core code fetch or a core dispatch-table read is being executed

- On reset, the CR16C stores the contents of the PSR in R2, bits 23 through 17 of the PC in R1, and bits 16 through 1 of the PC in R0.

4.1.6 Hardware Debug Mode

The CR16C provides support for an external hardware debug interface. In this hardware debug mode, the on-chip debug registers and external debug signals allow the external debug module to set watchpoints or stall the processor and access all of the processor registers.

If the hardware debug mode is enabled, the core cannot access the debug control registers. The contents of the PSR.T and PSR.P bits is ignored, i.e., writes to these bits are allowed and reads return an undefined result.

4.1.7 Debug Control and Status Registers

The debug control and status registers can be modified and read via LPR/SPR and LPRD/SPRD instructions. If hardware debug mode is enabled, a read from these registers results in 0x0000 and a write to these registers is ignored.

Note: If no Hardware Debug Module is present, the Debug Control and Status registers are reset on CR16C reset. Otherwise, the Debug Control and Status Registers are reset on a debug reset; refer to the description of the Hardware Debug module.

Debug Base Register (DBS)

The DBS Register selects the debug register pair and holds the status of all debug channels. The DBS Register is always accessible, independent of the DBS.BANK settings. The format of the DBS is shown in Figure 4-8.

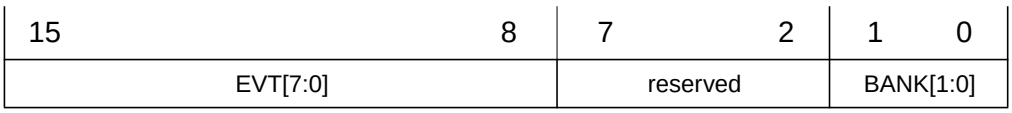


Figure 4-8. Debug Base Register (DBS)

BANK **Debug BANK Select**. This bit selects the currently accessible Debug Register Bank. A pair of debug channels consisting of CARn, CARn+1 DCR and DSR is mapped into the core debug register space, accessible via LPR/SPR instructions. The debug registers selected are calculated by n = BANK * 2.

EVT **Debug EVenT.** There is one EVT bit for every debug channel. This bit is a logic OR of the bits DSRb.BRDn, DSRb.BWRn and DSRb.BPCn bits. For debug channel 0, the DSR0.EXF bit is also part of the equation. The EVT bits are cleared when all corresponding source bits are cleared.

Compare Address Registers (CARn)

The CARn holds the compare address value for debug channel n. The format of the CARn is shown in Figure 4-9.

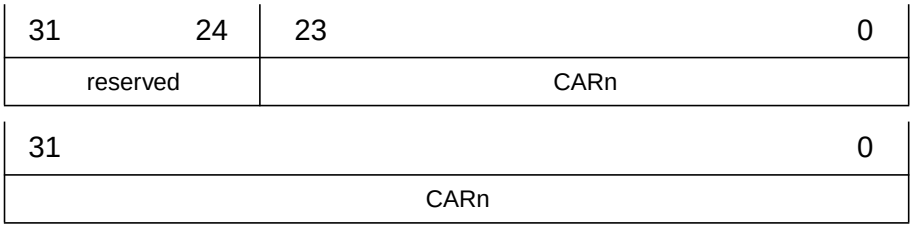


Figure 4-9. Compare Address Register (CARn)

Debug Control Register (DCRb)

The DCR controls the compare-address match, external tag on fetch, PC match debug for a set of two debug channels. The DCR of debug channel 0 also controls the alternate behavior of the BPT, TRC, ISE and DEBG breakpoints. 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.

Exception addressing for ISEs When address range matching is enabled (DCR0.EAR), the configuration of the address range matching is provided in the configuration bits for comparator 0: PC, CWR, CRD, and DEN. The CAR mask bits are independent of the address range match enable.

Address-match with bit-manipulation instructions Using the compare-address match for memory locations accessed by the **CBIT**, **SBIT**, and **TBIT** instructions requires special care. If both read and write breakpoints are enabled (both **DCR.CRD** and **DCR.CWR** are set), only the read breakpoint is acknowledged, setting just the **DSR.BRD** status bit.

If an unaligned transaction is used to access a word variable (i.e., Addr0 = 1), only the byte containing the relevant bit to be tested and modified is read and/or written. DCR should be programmed to match the appropriate byte for a match breakpoint to occur.

The DCR is cleared on reset (SFDBGRSTP goes active high). The format of DCRL and DCRH is shown in Figures 4-10 and 4-11.

15	14	13	12	11	10	9 8	7	6	5	4	3 2	1 0
DEN1	CRD1	CWR1	PC1	WPE1	ESQ	MSK1	DEN0	CRD0	CWR0	PC0	WPE0 EAR	MSK0

Figure 4-10. Debug Control Register L (DCRL)

31 24	23	22	21	20	19	18	17	16
reserved	DBGL	res	AISE	ADBG	ATRC	ABPT	res	EFB

Figure 4-11. Debug Control Register H (DCRH)

MSKn **Address Mask debug channel n.** The MSKn[1:0] bits select a mask for the CARn registers, according to Table 4-1.

Table 4-1. Address Mask Bit Function

MSKn		Function
Bit 1	Bit 0	
0	0	no mask
0	1	mask CARn[0]
1	0	mask CARn[1:0]
1	1	mask CARn[2:0]

These are valid for both compare-address matching and during PC-match. In addition, the masks are functional when the EAR bit is set.

EAR **Enable Address Range Breakpoint**. Enables breaking on a PC or address match in the range between CARn and CARn+1 within the same debug register block (b). CARn+1 >= range >= CARn. When set, configuration parameters for CARn are used for the range breakpoint.

ESQ **Enable Complex (Sequential) Breakpoint**. Enables a complex breakpoint using the CARn and CARn+1 of the same debug block in sequence. A breakpoint is executed if the condition for breakpoint n+1 occurs after the condition for breakpoint n.

WPEn **Watchpoint Enable debug channel n**. This bit is only available in hardware debug mode. When set, enables generation debug event outputs on detection of a debug event.

PCn **PC Match** or **Compare-Address Match channel n**. When set, enables generation of a DBG trap on a PC match on the corresponding CAR. When clear, enables generation of a DBG trap on a compare-address match.

CWRn **Compare-Address on Write channel n**. When set, enables address comparison for write operations.

CRDn **Compare-Address on Read channel n**. When set, enables address comparison for read operation.

DENn **Debug Condition Enable channel n**. Enables either the compare-address match or PC match condition, depending on the value of the PC bit for the corresponding CAR. Both DEN0 and DEN1 are cleared when a PC match occurs and DSR.BPC is set, or when address-match is achieved, and either DSR.BRD or DSR.BWR is set. These bits are also cleared when an externally tagged instruction reaches execution and DSR.EXF is set.

EFB **Enable Tag-at-Fetch Breakpoint**. Enables breaking by an external device on an instruction tagged during the fetch phase. EFB is cleared when the tagged instruction reaches execution, and DSR.EXF is set. This bit is also cleared when a PC match occurs and DSR.BPC is set, or when address-match is achieved, and either DSR.BRD or DSR.BWR is set.

This bit is only available in DCR0.

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

This bit is only available in DCR0.

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

This bit is only available in DCR0.

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

This bit is only available in DCR0.

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

This bit is only available in DCR0.

DBGL **Debug-In-System-Emulation Support Lock**. A set-once bit (cleared only by reset) that locks 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.

This bit is only available in DCR0.

Debug Status Register (DSRb)

The DSRb indicates debug and breakpoint conditions that have been detected for the breakpoint controlled by debug block b. When the CPU detects an enabled debug condition, it sets the appropriate bits in the DSRb to 1. Bits 0 through 23 of the DSR are cleared to 0 at reset. In addition, software must clear all the bits in the DSR when appropriate.When address range matching is enabled (DCRb.EAR), the status of the address range matching is given in the status bits for comparator n - BPCn, BWRn and BRDn (not by the status of comparator n+1).

The DSR is cleared on reset. The format of the DSR is shown in Figure 4-12.

15 7	6	5	4	3	2	1	0
reserved	BRD1	BWR1	BPC1	EXF	BRD0	BWR0	BPC0

Figure 4-12. Debug Status Register (DSRb)

BPCn **Program Counter channel n**. Set when a PC match is detected for the corresponding CARn.

BWRn **Write channel n**. Set when a compare-address match is detected for a data write for the corresponding CARn.

BRDn **Read channel n**. Set when a compare-address match is detected for a data read for the corresponding CARn.

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

This bit is only available in DSR0.

4.2 CACHE SUPPORT

The CR16C supports integration of an Instruction Cache (IC) and a Data Cache (DC), although the cache itself is not provided as part of the core.

The contents of the IC and DC are automatically loaded by the CR16C to maintain copies of recently used instructions and data values. The contents of the IC and DC can also be locked to hold copies of selected memory locations. This section describes the organization and operation of the IC and DC support.

The configuration register contains configuration bits to enable an IC and a DC. In addition, lines within the cache may be locked.

Instructions are provided to invalidate either or both caches, and to determine if the entire cache or only unlocked lines should be invalidated.

4.2.1 Instruction Cache Operation

The IC is enabled for an instruction fetch whenever the CFG.IC bit is 1. IC operation under various conditions is described below.

IC disabled If the IC is disabled, then the instruction fetch bypasses the IC, and the contents of the IC are unaffected. Instructions are read directly from external memory, and are transferred to the instruction queue through the IC. If the IC is not present, the main data bus is connected to the instruction cache data bus directly. If an IC is present but is currently disabled, these signals are connected together through the cache.

IC enabled If the IC is enabled for an instruction fetch, then bits of the address of the instruction in the PC are decoded to select the set of cache lines where the instruction may be stored. The selected set is read, and the tags and validity bits are compared with the relevant bits of the address of the instruction. The lower bits of the address in the PC identify the word and the corresponding validity bit in each line of the selected set.

IC hit If one of the tags matches and the selected word is valid, then the instruction is transferred directly to the instruction queue for execution. Otherwise, the missing data is read from external memory, as described below.

IC miss, lines locked If the instruction is not in the IC and if the contents of all lines in the set are locked, the IC is considered disabled and data from the main bus is used.

IC miss, at least one line not locked

If the instruction is not in the IC, and if at least one of the lines in the set is not locked, then:

- If the tag of either line in the set matches the address in the PC, then that line is selected for updating.
- If neither tag matches, *and*
  - at least one of the lines in the set is not locked, then the unlocked line is selected for replacement.
  - both lines are unlocked, then the least recently used line is selected for replacement.

Any missing data is read from external memory and transferred to the instruction queue for execution, regardless of whether the selected line is replaced or merely updated.

Instruction prefetching

After the CR16C has completed fetching a missing instruction from external memory, the IC may continue to prefetch sequential words in the same line. The memory from which the data is fetched can limit the number of words that can be prefetched.

The IC stops prefetching under any of the following conditions:

- When memory indicates that the maximum number of words has been prefetched
- When the end of the IC line is reached
- When a non-sequential fetch is issued or
- When a cache invalidation request is issued.

Whenever the cache is invalidated by a **CINV** instruction, the SFICIVP signal is asserted. In addition, the SFSELINVP signal is asserted if only the unlocked lines need to be invalidated.

4.2.2 Instruction Cache Invalidation

During execution of self-modifying code, it may be necessary to invalidate the IC. When a locked line is invalidated, it also becomes unlocked.

The contents of the IC can be invalidated by software or hardware.

Software invalidates the cache as follows:

- Clearing the CFG.IC bit invalidates the entire IC contents, including locked lines, and initializes a LRU selection bit to 0.
- Executing the **CINV** instruction invalidates either the entire IC, or if the U option is used, only unlocked lines.

The IC module may also provide support for invalidation via direct input.

4.2.3 Data Cache Operation

A Data Cache (DC) may be present on the core bus to provide faster access for more frequently used data. The DC is enabled for a data read if CFG.DC is 1 and the DC access attempt was not caused by an interrupt-acknowledge bus cycle.

DC disabled If the DC is disabled, then the data read bypasses the DC, and the contents of the DC are unaffected. The data that is read from external memory is used to execute the instruction.

DC enabled If the DC is enabled for a data read, then bits of the address of the data are decoded to select the set of lines where the instruction may be stored. The selected set is read and the tags are compared with the most significant bits of the address; the lower bits are used to determine the corresponding validity bit in each line of the selected set.

DC hit If one of the tags matches and the word is valid, then the data is used to execute the instruction. Otherwise, the missing data is read from external memory, as described below.

DC miss, lines locked If the data is not in the DC, and if the contents of both lines in the set are locked to fixed locations that do not match the address of the data, then the DC is considered disabled and data is read from an external memory.

DC miss, at least one line not locked If the data is not in the DC, and if at least one of the lines in the set is not locked, then:

- If the tag of a line in the set matches the address in the data, then that line is selected for updating.
- If no tag matches, *and*
  - At least one of the lines in the set is not locked, then the unlocked line is selected for replacement.
  - All lines are unlocked, then the least recently used line is selected for replacement.

Any missing data is read from external memory and used to execute the instruction, regardless of whether the selected line is being replaced or merely updated.

4.2.4 Data Write Operation

The DC is enabled for a data write whenever CFG.DC is 1.

The lower address bits of the data address are decoded to determine the set of lines where the data may be written. The lines of the selected set are read and both tags are compared with the most significant bits of the data's address.

Several bytes may be written to the cache, depending on the operand's length. Some of the least significant bits of the data's address select the appropriate word and the corresponding validity bit in each line of the selected set.

If the data is not located in the DC, then the contents of the DC are unaffected. In addition, the data may be written into the cache in a new line, if available. The data is always written to external memory whether the DC is updated or not.

4.2.5 Data Cache Invalidation and Coherence Support

To maintain coherence between the DC and external memory, a request can be issued to invalidate the whole DC whenever a location in memory is modified by a bus master other than the CR16C itself. This technique is appropriate, for example, for a single processor system with a low rate of I/O transfers to memory. When a locked line is invalidated, it also becomes unlocked.

The contents of the DC can be invalidated by software or hardware.

Software invalidates the DC as follows:

- Clearing the CFG.DC bit invalidates the entire contents of the DC, including locked lines, and initializes the LRU selection bit to 0.
- Executing the **CINV** instruction invalidates either the entire DC, or if the U option is used, only unlocked lines.

The DC module may also provide support for invalidation via direct input.

4.2.6 Data Cache Monitoring

Bus status codes and some data cache signals can be used to maintain an external copy of the valid contents of the on-chip DC.

When the data cache is disabled (SFDCENP is low), all its entries are invalid and all locked bits are cleared.

Bus status codes indicate when a data read operation from external memory is being executed (see Section 7.8.1 on page 7-9).

Whenever the cache is invalidated by a **CINV** instruction, the SFDCIVP signal is asserted. In addition, the SFSELINVP signal is asserted if only the unlocked lines need to be invalidated.

4.3 INSTRUCTION EXECUTION ORDER

The CR16C 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 4-13 shows these states and the transitions between them.

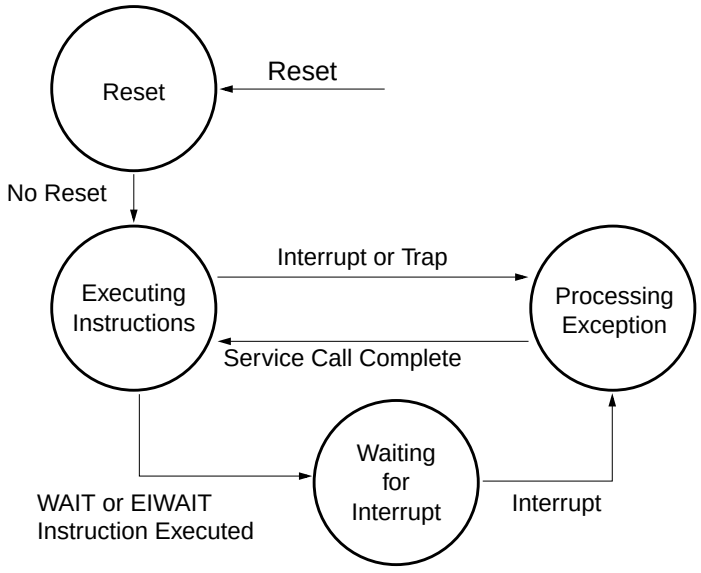


Figure 4-13. CR16C Operating States

Reset

When the reset input signal is activated, the CR16C enters the reset state. In this state, the contents of certain dedicated registers are initialized, as detailed in Section 3.5 on page 3-13.

Executing instructions

When the reset signal is deactivated, the CR16C enters the executing-instructions state. In this state, the CR16C executes instructions repeatedly until an exception is recognized, a **WAIT** instruction is executed or an **EIWAIT** instruction is executed.

Processing exception

When an exception is recognized, the CR16C enters the processing exception state in which it saves the PC and the PSR contents. The processor then reads an absolute code-address from the interrupt dispatch table and branches to the appropriate exception service procedure. Refer to Section 3.4 on page 3-4 for more information.

To process maskable interrupts, the CR16C also reads a vector value from an Interrupt Control Unit (ICU).

After successfully completing all data references required to process an exception, the CR16C reverts to the executing instructions state.

Waiting for interrupt

When a **WAIT** or an **EIWAIT** instruction is executed, the CR16C enters the waiting for interrupt state and becomes idle. When an interrupt is detected, the processor enters the processing exception state.

4.3.1 The Instruction Pipeline

The CR16C can overlap operations for several instructions, using a pipelined technique to enhance its performance. While the CR16C is fetching one instruction, it can simultaneously be decoding a second instruction and calculating results for a third instruction. See Figure 4-14.

In most cases, pipelined instruction execution improves performance while producing the same results as strict, sequential instruction execution. Under certain circumstances, the effects of this performance enhancement are visible to system software and hardware as differences in the order of memory references performed by the CR16C.

Instruction fetches

The CR16C 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 from previous instructions have been completed. Figure 4-14 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 and between instructions.

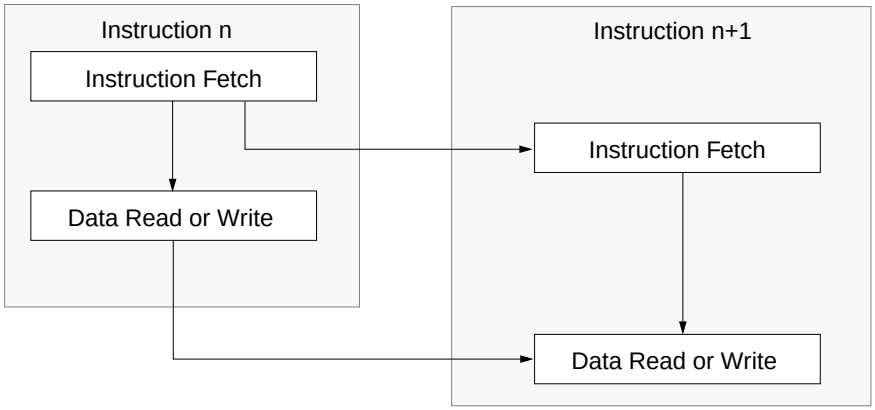


Figure 4-14. Memory References for Consecutive Instructions

Overlapping operations As a consequence of overlapping operations for several instructions, the CR16C may fetch an instruction but not execute it (e.g., if the previous instruction causes a trap). The CR16C reads source operands and writes destination operands for executed instructions only.

Dependencies The CR16C does not check for dependencies between fetching the next instruction and writing the results of the previous instructions. Special care is therefore required when executing self-modifying code.

4.3.2 Serializing Operations

The CR16C serializes instruction execution after processing an exception. The results of all preceding instructions are written to a destination before the first instruction of the exception service procedure is fetched. This fetch is considered non-sequential.

The CR16C also serializes instruction execution after executing the following instructions: **LPR**, **RETX**, and **EXCP**.

Whenever the core serializes execution, the instruction pipeline is flushed and the processor restarts filling it by fetching new instructions from the next memory location. This location is pointed to by the updated program counter (PC). Instruction execution begins only when the first fetched instruction rolls through the pipeline and reaches the execution stage. For more information, see Appendix A.

Chapter 5
INSTRUCTION SET

This chapter describes in details each of the CR16C instructions.

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), *procreg* (processor register), *imm* (immediate), *abs* (absolute) *rel* (relative), or *idx* (index relative)The reg and rel addressing modes are followed by a specifier *rp* (register pair) or *r* (register). The imm addressing mode is followed by a maximum bit size specifier. Many instructions support multiple imm sizes for efficient code memory usage. For more details, see Section “ADDRESSING MODES” on page 2-15 and Appendix B.2.1 on page B-3 for definitions of the imm formats.

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.

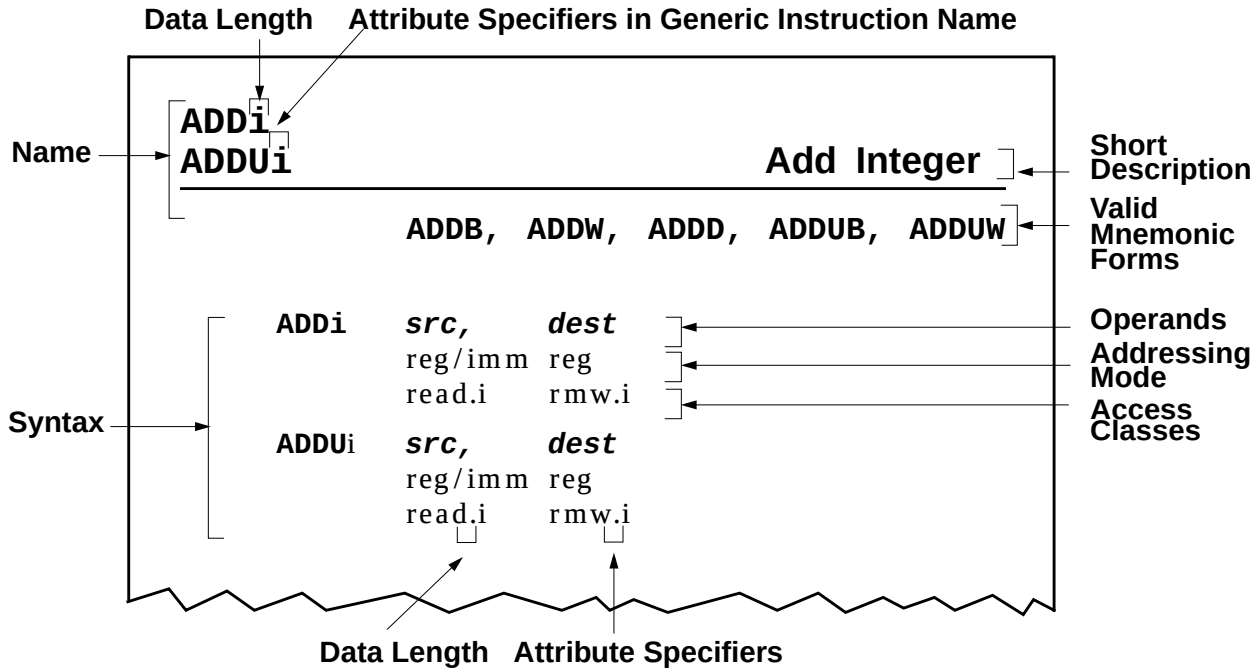


Figure 5-1. Instruction Header Format

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 **B** (byte), or **W** (word) in the actual instruction name. In the access class, the **L** (long) specifier stands for the long, 24-bit address/displacement.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 in gray. See Figure 5-2.

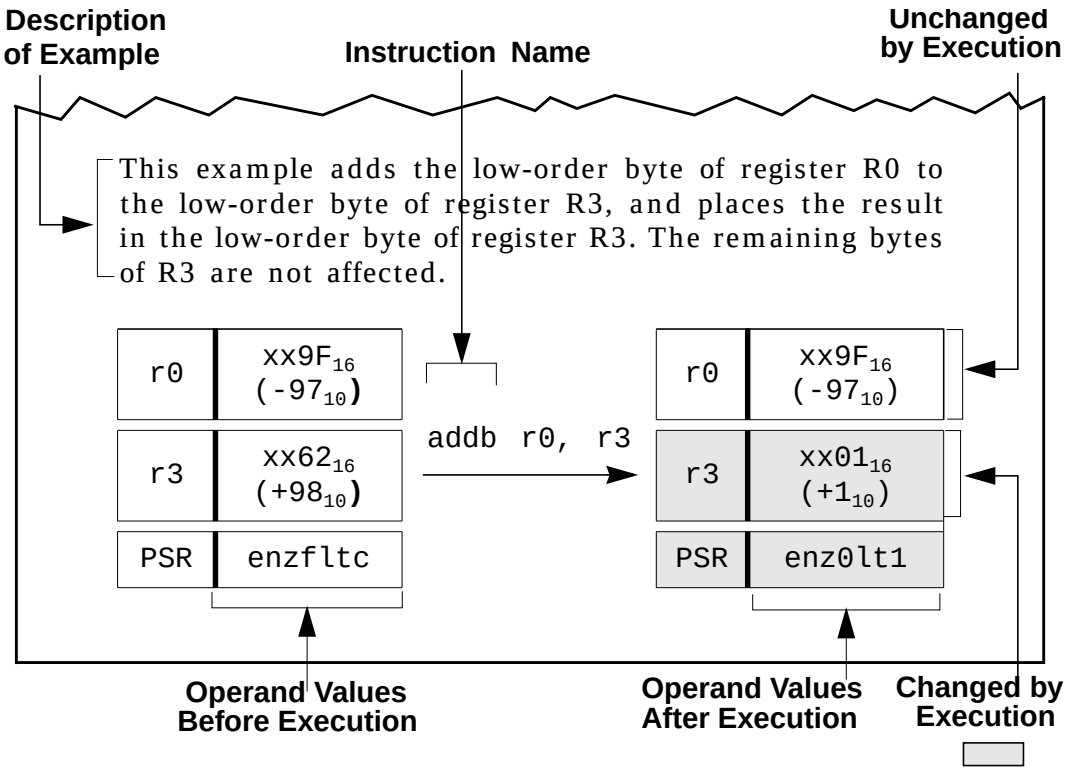


Figure 5-2. Instruction Example Format

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 instruction set is described in detail in the following pages.

ADDB, ADDW, ADDD, ADDUB, ADDUW

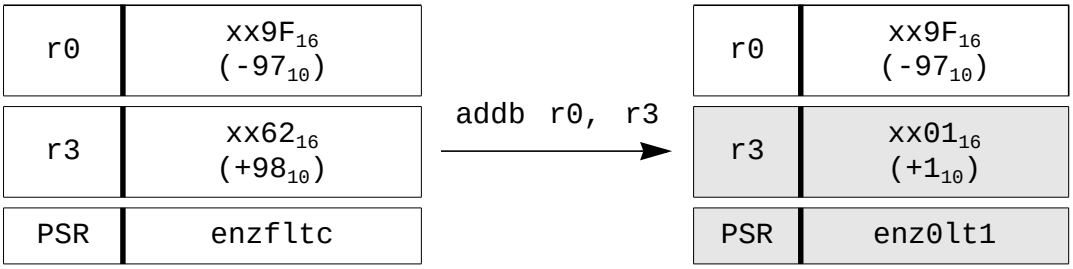
ADDi	***src***,	***dest***
	reg.r/imm4/16	reg.r
	read.i	rmw.i
ADDD	***src***,	***dest***
	reg.rp/imm4/16/20/32	reg.rp
	read.D	rmw.D
ADDUi	***src***,	***dest***
	reg.r/imm4/16	reg.r
	read.i	rmw.i

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

Flag During execution of an **ADDi or ADDD** 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.

Trap 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.



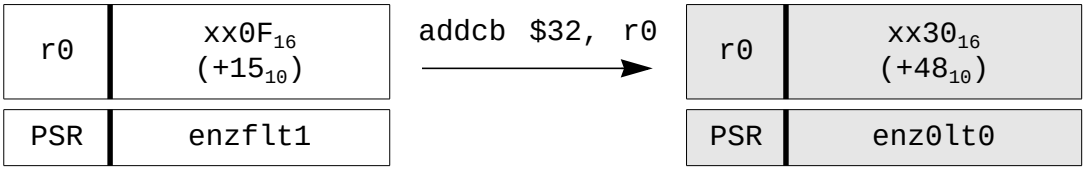
ADDCi	***src***,	***dest***
	reg/imm4/16	reg
	read.i	rmw.i

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

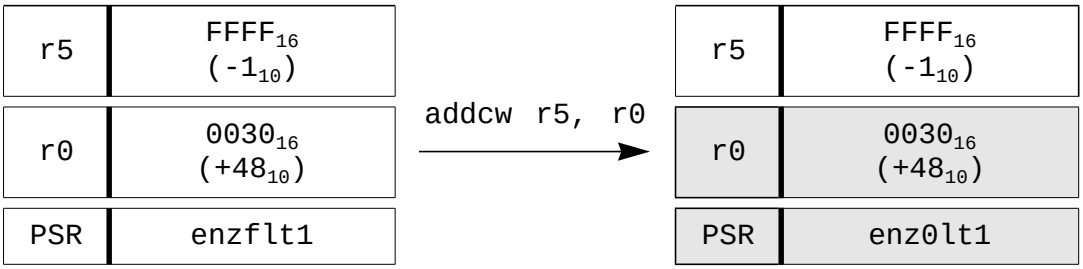
Flag PSR.C is set to 1 if a carry occurs, and cleared to 0 if there is no carry. PSR.F is set to 1 if an overflow occurs, and cleared to 0 if there is no overflow.

Trap None

Example 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	***src***,	***dest***
	reg.r/imm4/16	reg.r
	read.i	rmw.i
ANDD	***src***,	***dest***
	reg.rp/imm32	reg.rp
	read.D	rmw.D

The **ANDi/ANDD** instruction performs a bitwise logical AND operation on the ***src*** and ***dest*** operands, and places the result in the ***dest*** operand.

Flag

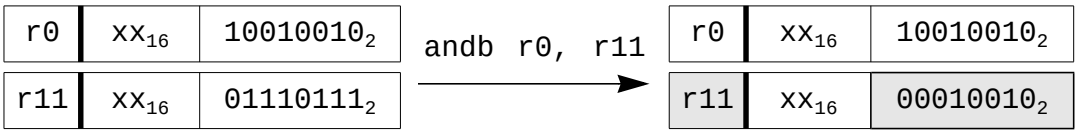
None

Trap

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, ASHUD

ASHUi ***count***, ***dest***
reg.r/imm4/5 reg.r
read.B rmw.i

ASHUD ***count***, ***dest***
reg.r/imm6 reg.rp
read.B rmw.D

The **ASHUi** instruction performs 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; in the range –15 to +15 if **ASHUW** is used; and in the range –31 to +31 if **ASHUD** 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***.

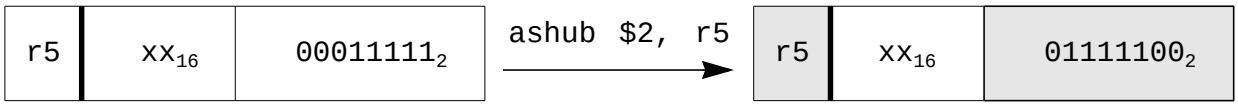
Note that for ASHUD, if the ***dest*** operand consists of two registers (***dest***+1,***dest***), then in addition to what is described above for all ASHUi instructions, shifts have the following effects:

- A shift to the left causes high-order bits to be shifted out of ***dest*** register to ***dest***+1 register.
- A shift to the right causes low-order bits of ***dest***+1 register to be shifted out to ***dest*** register.

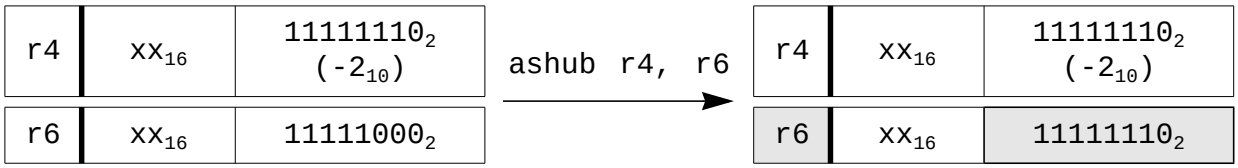
Flag None

Trap None

Example 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.



BAL	***link***,	***dest***
	reg.rp	disp25
	write.L	disp

The address (bits 23 to 1) of the next sequential instruction is first stored in the ***link*** operand. Then, program execution continues at the address specified by ***dest***, sign extended to 25 bits, plus the current contents of the PC register. as follows: PC <- (PC+ sext_25(disp)). The result is stored in PC before being used.

If the resulting PC value is less than 0x00_0000 or greater than 0xFF_FFFF, this instruction causes an IAD trap.

If the link operand is (ra), with CFG.SR=0, or (era,ra), with CFG.SR=1, the instruction size is 4 bytes; otherwise, it is 6 bytes.

Flag

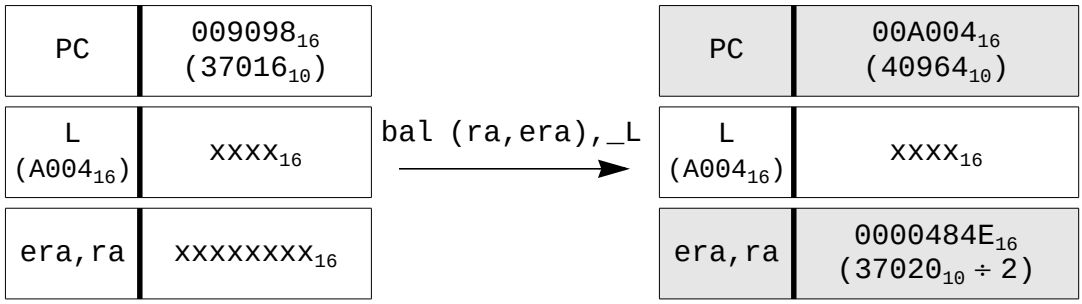
None

Trap

None

Example

Saves bits 23 through 1 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.



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

Bcond ***dest***

disp

disp

Table 5-1. If the condition specified by ***cond*** is true, the **Bcond** instruction causes a branch in program execution. Program execution continues at the location specified by ***dest***, sign extended to 25 bits, plus the current contents of the Program Counter. If the condition is false, execution continues with the next sequential instruction. Table 5-1 summarizes the different addressing calculations.**BR/BRcond Target Addressing Methodology**

Displacement Size (Signed)	Inst Size	Address Range	Address Calculation
9 bits	2	0 - 16M	PC <- (PC + sign extend to 25(disp))
17 bits	4	0 - 16M	PC <- (PC + sign extend to 25(disp))
25 bits	6	0 - 16M	PC <- (PC + disp)

cond is a two-character condition code that describes the state of a flag, or flags, in the PSR register. If the flag(s) is/are set as required by the specified ***cond***, the condition is true; otherwise, the condition is false. Table 5-2 describes the possible ***cond*** codes and the related PSR flag settings.

Flag None

Trap None

Table 5-2. *cond* Codes and PSR Settings

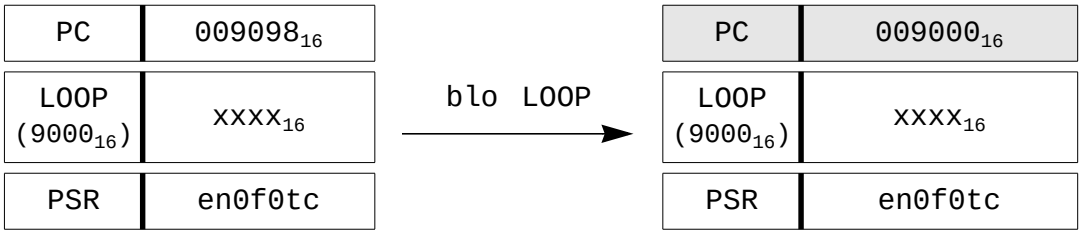
cond **Code**	**Condition**	**True State**
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal To	N flag is 0

The assembler encodes the displacement by dividing it by two. The CR16C expands this encoding by multiplying the value by two. The least significant bit of the displacement is always 0.

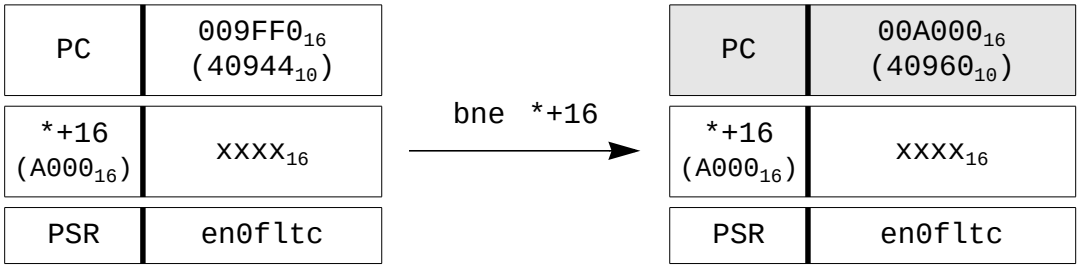
cond Code	Condition	True State
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal To	Z or N flag is 1

Examples

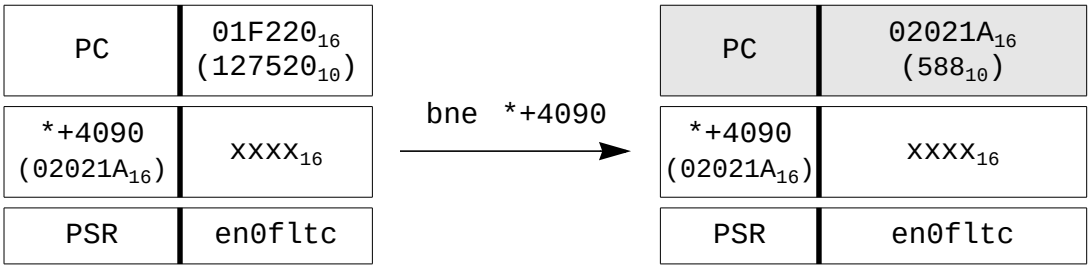
1. Passes execution control to the instruction labeled LOOP by adding $FFFF68_{16}$ to the PC register, if the PSR.Z and PSR.L flags are 0.



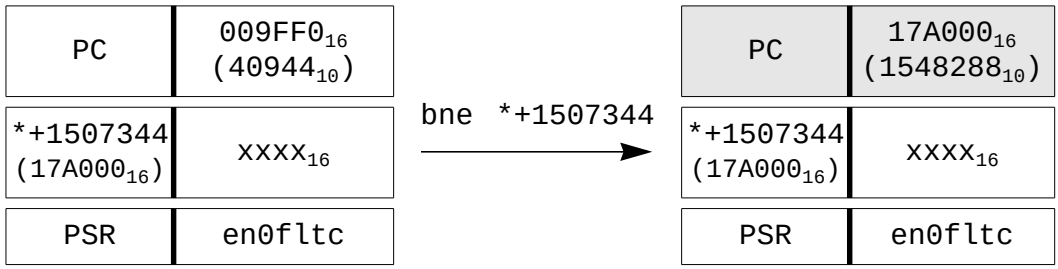
2. Passes execution control to a non-sequential instruction if the PSR.Z flag is 0. The instruction passes execution control by adding 16 to the PC register.



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.



4. 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.



BCond0i	***src***,	***dest***
	reg.r	disp5
	read.i	disp

The **BCond0i** instruction compares the signed contents of ***src*** to '0', and branches upon equality or non-equality (according to ***cond***) as shown in Table 5-3. The target address is determined by adding the 5-bit displacement (unsigned even 2-32) to the current value of the program counter. Only forward branching is supported.

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

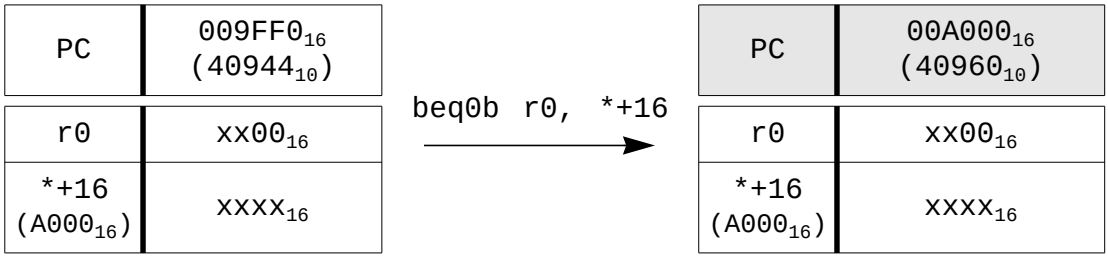
Table 5-3. Conditional Branch

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

Flag None

Trap None

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



The assembler encodes the displacement by dividing it by two and subtracting one. The CR16C reverts this encoding by adding one and the multiplying the value by two, the least significant bit of the displacement is always 0

BR ***dest***
disp 9/17/25
disp

dest is an even integer, sign extended to 25 bits and added to the current contents of the PC register. The result is loaded into the PC register. Program execution continues at the location specified by the updated PC register. Table 5-1 describes the detailed address calculation performed for the **BR** instruction.

Flag

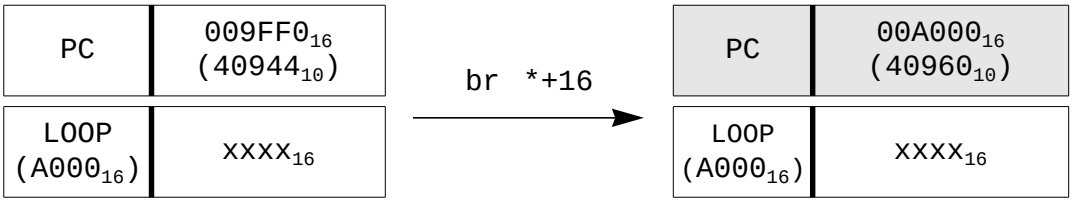
None

Trap

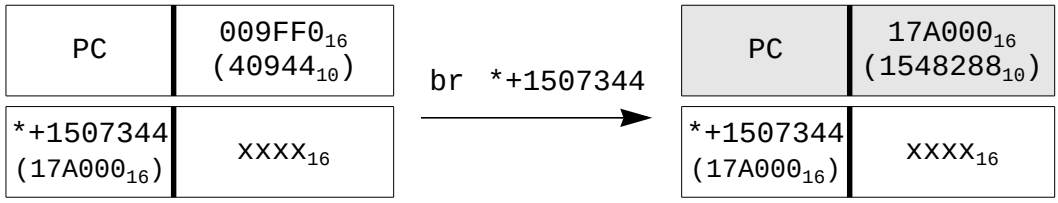
None

Examples

1. Passes execution control to the instruction at address *+16 by adding +16 to the PC register.



2. 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. The **Branch** command executes the same way as **BCond**, since the condition is effectively always true.

CBITB, CBITW

CBITi	***position***,	***dest***
	imm4	abs/rel.r/rel.rp/idx
	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.

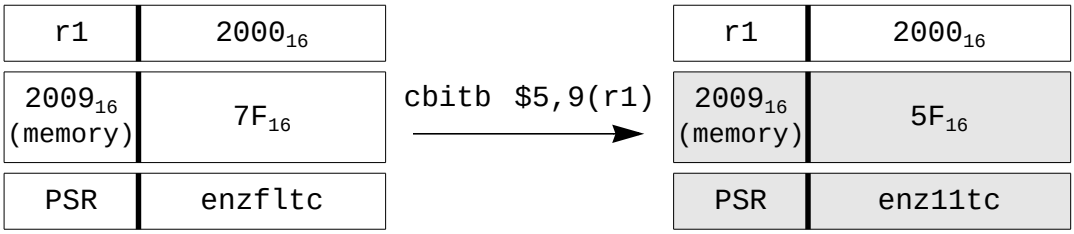
As there is no native support for clearing a bit in a double-length core register when CFG.SR=0, the ANDD instruction should be used.

See Table 5-4 for addressing modes.

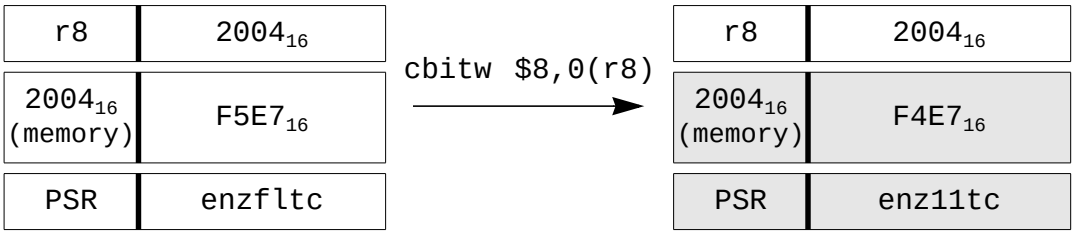
Flag Before the specified bit is modified, its value is stored in PSR.F.

Trap None

Examples 1. Clears bit in position 5 in a byte operand at 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}.

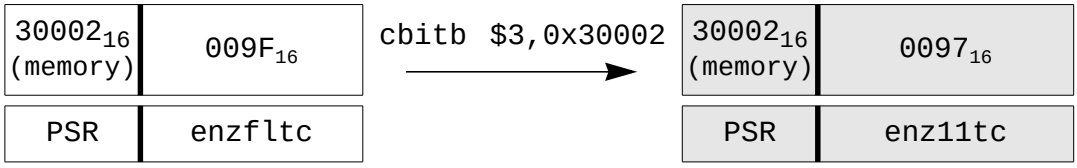


Table 5-4. CBIT/SBIT/TBIT and STOR Imm Addressing Methodology

Addressing Format	Mode	Displace-ment Range	Instr Length (Byte)	Address range	Address Calculation
0(RPbase)	rel.rp	0	2	16M	Rpbase
disp16(RPbase)	rel.rp	0 to 64K – 1	4	16M	Rbase + zext24(disp16)
disp20(RPbase)	rel.rp	0 to 1M – 1	6	16M	Rbase + zext24(disp20)
[Rindex]disp14(RPbasex[a])[b]	idx	0 to 16K – 1	4	16M	Rindex + (RPbasex) + zext24(disp14)
disp14(Rbase)[c]	rel	0 to 16K – 1	4	64K+16K	Rbase + zext24(disp14)
[Rindex]disp20(RPbasex[a])[b]	idx	0 to 1M – 1	6	16M	Rindex + (RPbasex) + zext24(disp20)
disp20(Rbase)	rel	0 to 1M – 1	6	1M	Rbase + zext24(disp20)
[Rindex]abs20[b]	idx	0 to 1M - 1	4	16M	Rindex + zext24(disp20)
abs20	abs	0 to 1M - 1[d]	4	1M	zext24(abs20) | remap[e]
abs24	abs	0 to 16M -1	6	16M	abs24

a. RPbasex - Base register pair for relative addressing only: (R1,R0), (R3,R2), (R5,R4), (R7,R6), (R9,R8), (R11,R10), (R4,R3), (R6,R5)
b. when CFG.SR = 0
c. when CFG.SR = 1
d. The 1M addressable range is split into (0x0 to 1M-64k) and (16M-64k to 16M)
e. If (abs20 > 0xEFFFF) the resulting address is logically ORed with 0xF00000 i.e. addresses from 1M-64k to 1M are re-mapped by the core to 16M-64k to 16M.

CINV ***options***

The **CINV** instruction invalidates the contents of the on-chip instruction cache and/or data cache. **CINV** can invalidate either the entire contents, or only the unlocked entries, of the on-chip caches.

options are specified by listing the letters ***I***, ***D***, and/or ***U***. These options are independent of one another. The ***I*** option invalidates the instruction cache; the ***D*** option invalidates the data cache. If both ***I*** and ***D*** are specified, then both caches are invalidated. The ***U*** option invalidates only the unlocked lines in the listed cache(s). If the ***U*** option is not specified, the entire cache/caches is/are invalidated.

Flag

None

Trap

None

Examples

1. Invalidates the unlocked lines in the instruction cache.

```
cinv [i,u]
```

2. Invalidates all lines in both the instruction cache and the data cache.

```
cinv [i,d]
```

3. Invalidates all unlocked lines in both the instruction cache and the data cache.

```
cinv [i,d,u]
```

CMPi ***src1***, ***src2***
reg.r/imm4/16 reg.r
read.i read.i

CMPD ***src1***, ***src2***
reg.rp/imm4/16/32 reg.rp
read.D read.D

The **CMPi/D** 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.

Flag

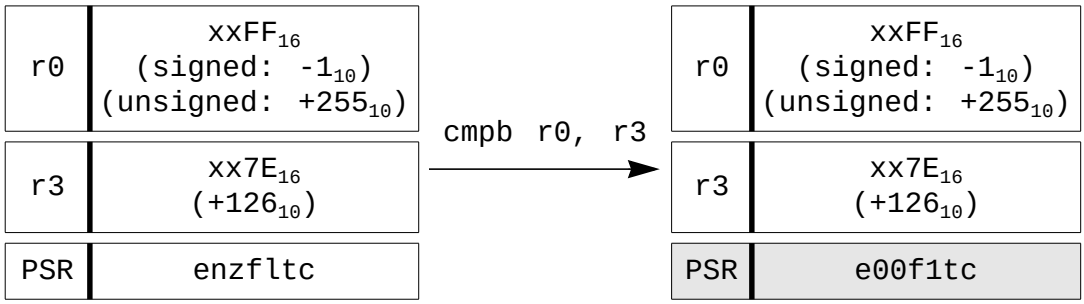
PSR.Z is set to 1 if ***src1*** equals ***src2***; otherwise it is cleared to 0. PSR.N is set to 1 if ***src1*** is greater than ***src2*** (signed comparison); otherwise it is cleared to 0. PSR.L is set to 1 if ***src1*** is greater than ***src2*** (unsigned comparison); otherwise it is cleared to 0.

Trap

None

Example

Compares low-order bytes in registers R0 and R3.



DI

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

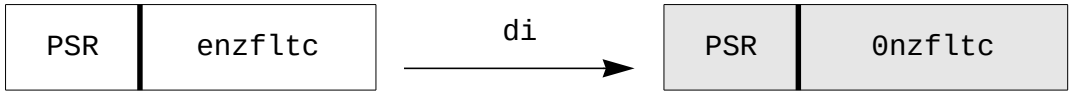
Note: The E flag in the PSR takes two cycles to update as a result of an **EI**, **EIWAIT** or **DI** instruction. Therefore, an exception occurring immediately after the **DI** instruction might still be recognized, and an exception immediately following **EI**/**EIWAIT** might be recognized only the next cycle, if it still is asserted.

In order to avoid any ambiguity it is advisable to have a disable_interrupt function which consists of a DI followed by a NOP.

Flag PSR.E is cleared to 0.

Trap None

Example Clears the PSR.E bit to 0.



EI

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

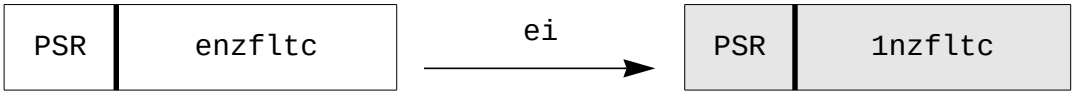
Note: The E flag in the PSR takes two cycles to update, as a result of an **EI**, **EIWAIT** or **DI** instructions Therefore, an exception occurring immediately after **DI** instruction might still be recognized, and an exception immediately following **EI**/**EIWAIT** might be recognized only the next cycle, if it still is asserted.

In order to avoid any ambiguity it is advisable to have a enable_interrupt function which consists of a EI followed by a NOP.

Flag PSR.E is set to 1.

Trap None

Example Sets the PSR.E bit to 1.



EIWAIT

The **EIWAIT** instruction suspends program execution until an interrupt occurs. This instruction also sets the PSR.E bit, enabling an 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.

Note: The E flag in the PSR takes two cycles to update, as a result of either an **EI**, **EIWAIT** or **DI** instruction. An exception occurring immediately after a **DI** instruction might still be recognized, and an exception immediately following **EI**/**EIWAIT** might be recognized only the next cycle, if it still is asserted.

Flag Sets the PSR.E bit to 1

Trap None

Example

```
eiwait
```

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.

Flag None

Trap The traps that occur are determined by the value of the ***vector*** operand, as shown in Table 5-5.

Table 5-5. Exception Traps

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

a. If any other vector is encoded, a UND trap is called; this includes DBG/ISE/IAD/TRC

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.rp
addr.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 by loading register bits 22 through 0 into bits 23 through 1 of the PC register. Bit 0 of the PC is 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 is/are set as required by the specified ***cond***, the condition is true; otherwise, the condition is false. Table 5-6 describes the possible ***cond*** codes and the related PSR flag settings:

Table 5-6. *cond* Codes and Related Flags

***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 To	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 To	Z or N flag is 1

Flag None

Trap None

Example Loads the address held in R3, R4 into the bits 1 through 23 of the PC register. Program execution continues at that address, if the PSR.Z and PSR.L flags are 0.

r3	1004_{16}
r4	$000c_{16}$
PC	$0A9098_{16}$
PSR	en0f0tc

jlo (r4,r3) →

r3	1004_{16}
	$000c_{16}$
PC	182008_{16} ($C1004_{16} \times 2$)
PSR	en0f0tc

JAL

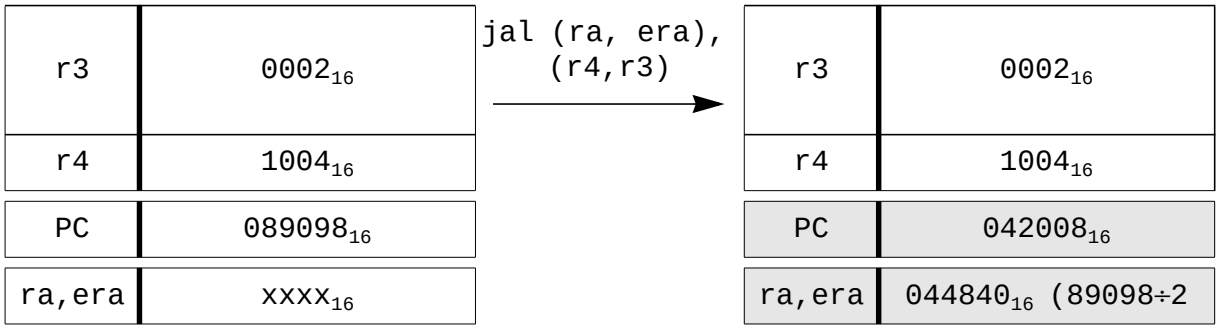
JAL ***link***, ***dest***
reg.rp reg.rp
write.L addr.L

Program execution continues at the address specified in the ***dest*** register, by loading register bits 22 through 0 into bits 23 through 1 of the PC register. Bit 0 of the PC register is cleared to 0. Bits 23 through 1 of the address of the next sequential instruction are stored in the ***link*** operand. If the link operand is (ra) when CFG.SR=0, or (era,ra) when CFG.SR=1, the instruction size is 2 bytes; otherwise, it is 6 bytes.

Flag None

Trap None

Example loads the address held in R3 into the PC register. Program execution continues at that address. The address of the next sequential instruction is stored in register RA.



JUMP ***dest***
reg.rp
addr.L

JUSR ***dest***
reg.rp
addr.L

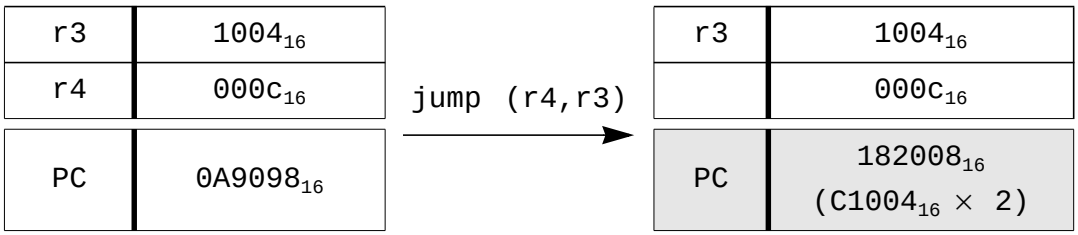
Program execution continues at the address specified in the ***dest*** operand , by loading the operand's bits 22 through 0 into bits 23 through 1 of the PC register. Bit 0 of the PC register is cleared to 0.

JUSR continues program execution in user mode. This is indicated by setting the PSR.U bit in addition to the JUMP.

Flag None for JUMP, but PSR.U is set for JUSR instruction.

Trap None

Example loads the address held in R4, R3 into bits 23 through 1 of the PC register. Program execution continues at that address.



LOADi ***src***, ***dest***
abs/rel.r/rel.rp/idx reg
read.i write.i

LOADD ***src***, ***dest***
abs/rel.r/rel.rp/idx reg.rp
read.D write.D

The **LOADi/LOADD** instruction loads the ***src*** operand from memory, and places it in the ***dest*** operand. Table 5-7 describes the addressing methodology for the instruction.

Table 5-7. LOAD/STOR Memory Addressing Methodology

Addressing Modes	Data Length	Mode	Displacement Range	Instr. Length (Byte)	Addr. Range	Address Calculation
disp4(RPbase)	b	rel.rp	0 to 13	2	16M	RPbase + zext24(disp4)
disp4(RPbase)	w/d	rel.rp	even numbers from 0 to 26	2	16M	RPbase + zext24(disp4)
disp16(RPbase)	b/w/d	rel.rp	0 to 64K - 1	4	16M	RPbase + zext24(disp16)
disp20(RPbase)	b/w/d	rel.rp	0 to 1M - 1	6	16M	RPbase + zext24(disp20)
-disp20(RPbase)[a]	b/w/d	rel.rp	-(1M - 1) to -1	6	16M	zext24(Rbase) + sext24(-disp20)
[Rindex]disp0(RPbasex[b])[c]	b/w/d	idx	0	2	16M	Rpbasex + Rindex
[Rindex]disp14(RPbasex[b])[c]	b/w/d	idx	0 to 16k–1	4	16M	RPbasex + Rindex + zext24(disp14)
[Rindex]disp20(RPbasex[b])[c]	b/w/d	idx	0 to 1M–1	6	16M	RPbasex + Rindex + zext24(disp20)
disp0(Rbase)[d]	b/w/d	rel.r	0	2	64K	zext24(Rbase)
disp14(Rbase)[d]	b/w/d	rel.r	0 to 16k–1	4	64K +16K	zext24(Rbase) + zext24(disp14)
disp20(Rbase)	b/w/d	rel.r	0 to 1M–1	6	1M +64K	zext24(Rbase) + zext24(disp20)
-disp20(Rbase)[a]	b/w/d	rel.r	-(1M - 1) to -1	6	1M	zext24(Rbase) + sext24(-disp20)

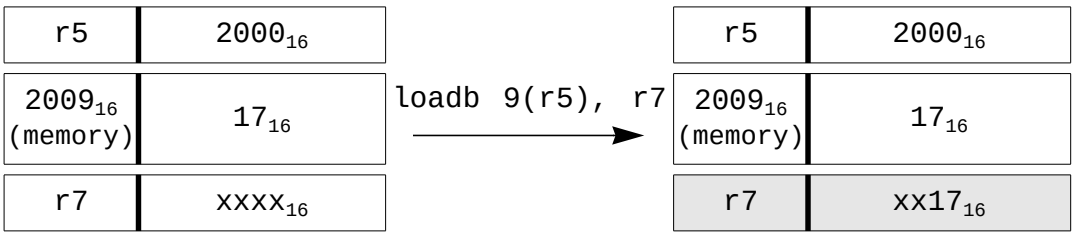
Addressing Modes	Data Length	Mode	Displacement Range	Instr. Length (Byte)	Addr. Range	Address Calculation
[Rindex]abs20[c]	b/w/d	idx	0 to 1M – 1	4	16M	Rindex + zext24(abs20)
abs20	b/w/d	abs	0 to 1M –1[e]	4	1M	zext24(abs20) | remap[f]
abs24	b/w/d	abs	0 to 16M - 1	6	16M	abs24

a. This displacement format may not be supported in future versions of the CPU. It is included for assembly level, backward compatibility with CR16B.
b. RPbasex - Base register pair for relative addressing only: (R1,R0), (R3,R2), (R5,R4), (R7,R6), (R9,R8), (R11,R10), (R4,R3), (R6,R5)
c. Supported when CFG.SR= 0
d. Supported when CFG.SR= 1
e. “The 1M addressable range is split into (0x0 to 1M-64k) and (16M-64k to 16M)”.
f. “If (abs20 > 0xEFFFF) the resulting address is logically ORed with 0xF00000 i.e. addresses from 1M-64k to 1M are re-mapped by the core to 16M-64k to 16M.”

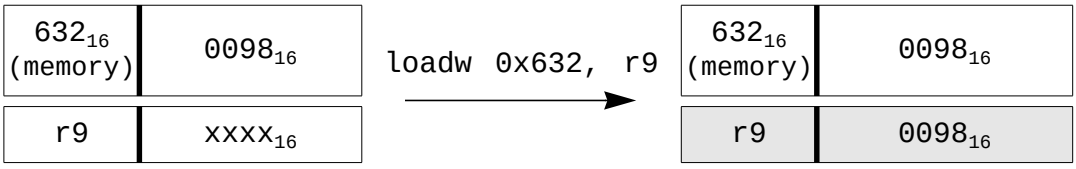
Flag None

Trap None

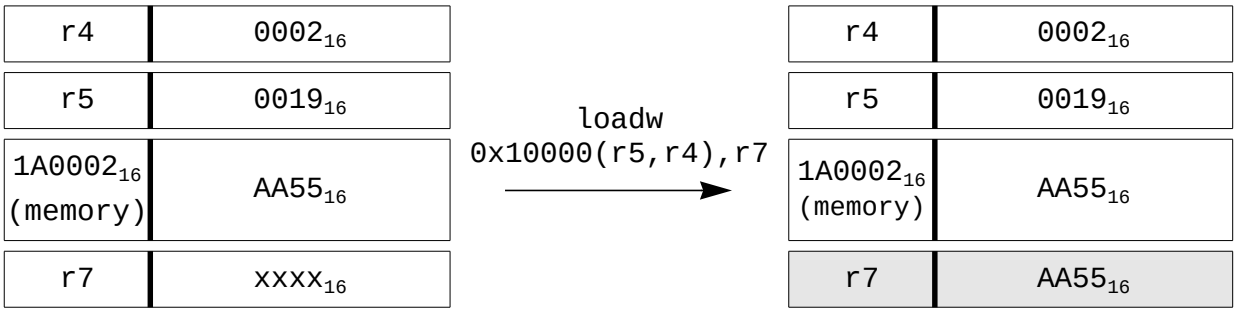
Examples 1. Loads a byte operand in address 9 (R5) to the low-order byte of register R7. The remaining byte of register R7 is unaffected.



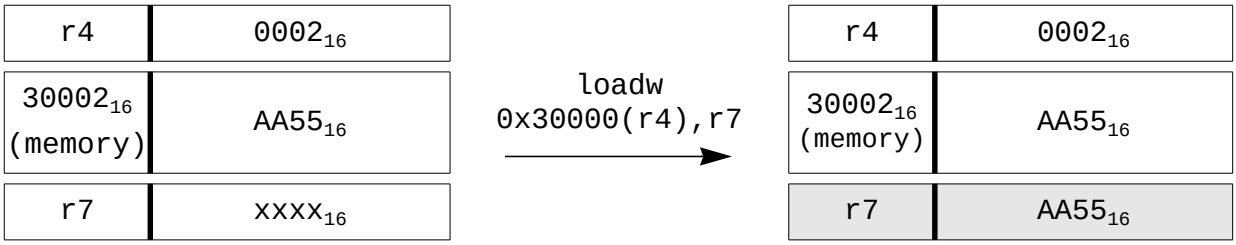
2. Loads a word operand in address 632 to register R9.



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 ***count***

imm3
read

LOADMP ***count***

imm3
read

The **LOADM** and **LOADMP** instructions load adjacent registers from memory. ***count*** reflects the total number of words to be loaded in the range 1 to 8. **LOADM** operates over the first 64K of memory and **LOADMP** operates over the entire memory range. The instruction always operates on a fixed set of registers as described below.

For **LOADM:**

- R0 contains the address of the first word in memory to be loaded;
- R2 is loaded with the lowest address word;
- R3 through R5 and then R8 through R11 are loaded from the next ***count***-1 consecutive addresses.

R0 is adjusted (incremented) by 2 for each word loaded, and therefore points to lower 16-bits of the next unread word in memory at the transfer-end. The address does not wrap around i.e., if R0 points to the end of the 64k addressable range LOADM overflows to addresses 0x010000 and following.

For **LOADMP:**

- (R1, R0) contains the address of the first word in memory to be loaded;
- R2 is loaded with the lowest address word;
- R3 through R5 and then R8 through R11 are loaded from the next ***count***-1 consecutive addresses.

(R1, 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 interruptible.

Flag None

Trap None

LPR	***src***,	***dest***
	reg.r	procreg
	read.W	write.W
LPRD	***src***,	***dest***
	reg.rp	procreg
	read.D	write.D

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

For the LPR instruction, if ***dest*** is ISPL or INTBASEL the least significant bit (bit 0) of the address is 0. If ***dest*** is INTBASEH, ISPH, USPH, CAR0H, or CAR1H bits 8 through 15 are always written as 0.

For the LPRD instruction, if ***dest*** is ISP or INTBASE the least significant bit (bit 0) of the address is 0. If ***dest*** is INTBASE, ISP, USP, CAR0, or CAR1 bits 24 through 31 are always written as 0.

On a LPRD of a 16-bit register, the upper 16 bits of the ***src*** are ignored.The processor registers in Table 5-8 may be loaded:

Table 5-8. Loadable Processor Registers

Register	LPR	LPRD
Processor Status Register	**PSR**	**PSR**
Configuration Register	**CFG**	**CFG**
Interrupt Base Register		**INTBASE**
Interrupt Base Low Register	**INTBASEL**	
Interrupt Base High Register	**INTBASEH**	
Interrupt Stack Pointer Register		**ISP**
Interrupt Stack Pointer Low Register	**ISPL**	
Interrupt Stack Pointer High Register	**ISPH**	
User Stack Pointer Register		**USP**
User Stack Pointer Low Register	**USPL**	
User Stack Pointer High Register	**USPH**	
Debug Status Register	**DSR**	**DSR**
Debug Condition Register		**DCR**
Debug Condition Low Register	**DCRL**	
Debug Condition High Register	**DCRH**	

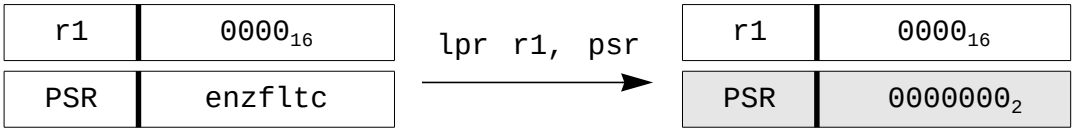
Register	LPR	LPRD
Compare Address 0 Register		**CAR0**
Compare Address 0 Register Low	**CAR0L**	
Compare Address 0 Register High	**CAR0H**	
Compare Address 1 Register		**CAR1**
Compare Address 1 Register Low	**CAR1L**	
Compare Address 1 Register High	**CAR1H**	

Refer to “REGISTER SET” on page 3-1 and to “INSTRUCTION SET” on page 6-1 for more information on these registers.

Flag PSR flags are affected by loaded values except for the U bit.

Trap When PSR.U is set, this instruction causes an UND trap.

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



LSHB, LSHW, LSHD

LSHi	***count,***	***dest***
	reg.r/imm4/5	reg.r
	read.B	write.i
LSHD	***count,***	***dest***
	reg.r/imm6	reg.rp
	read.B	write.D

The **LSHi/LSHD** 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, –15 to +15 if **LSHW** is used and –31 to +31 if **LSHD** 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.

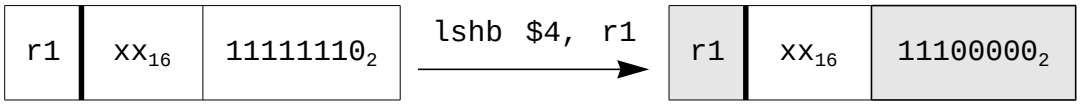
For LSHD, if the ***dest*** operand consists of two actual registers (***dest+1,dest***), in addition to what is described regarding all LSHi instructions, a shift to the left causes high-order bits to be shifted out of ***dest*** register to ***dest+1*** register, and a shift to the right causes low-order bits of ***dest+1*** register to be shifted out to ***dest*** register.

Note: The LSHi or LSHD instruction with a positive count is not natively supported by the core. It is mapped by the assembler to the corresponding ASHUi or ASHUD instruction, respectively.

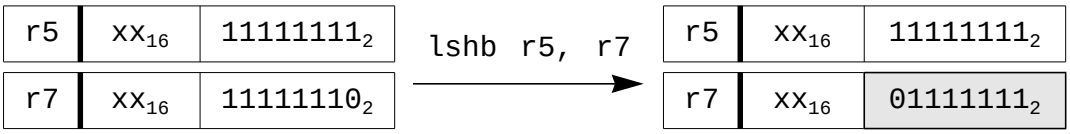
Flag None

Trap None

Examples 1. Shifts the low-order byte of register R1 four bit positions to the left. The remaining byte 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 byte of register R7 is unaffected.



MACSW	***src1***, ***src2***	***dest***
	reg.r, reg.r	reg.rp
	read.W	rmw.D

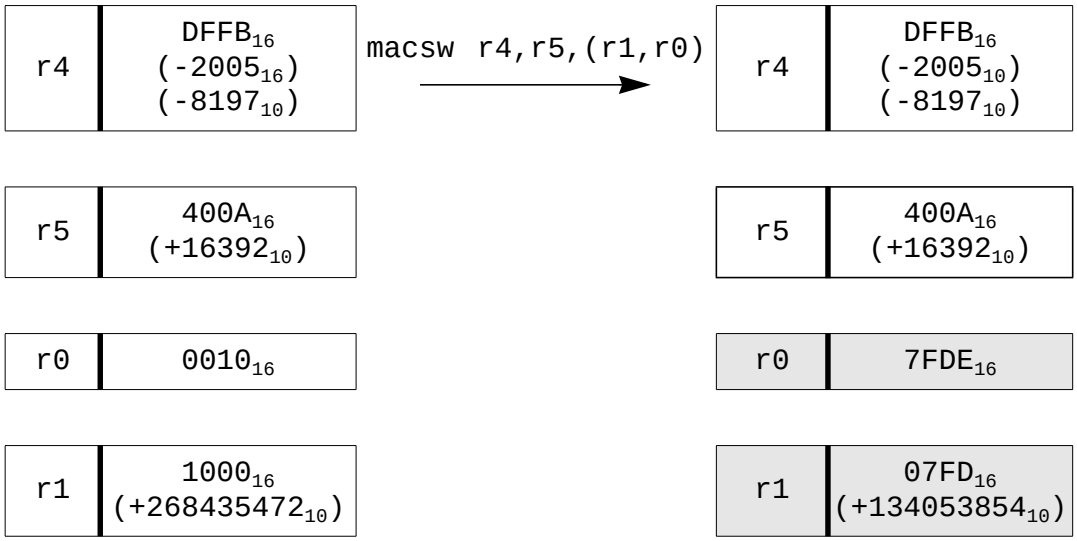
The **MACSW** instruction multiplies the ***src1*** operand by the ***src2*** operand and adds the result to the ***dest*** operand. The two src operands are interpreted as signed 16-bit integers. The result is viewed as a signed 32-bit integer.

During the addition, the result may overflow or underflow. In this case, the result is set (saturated) to the largest positive ($+2^{31}-1$) or largest negative (-2^{31}) number.

Flag None

Trap None

Example Multiplies register R4 by register R5, and adds the result to the register pair (R1,R0).



MACUW	***src1***, ***src2***	***dest***
	reg.r, reg.r	reg.rp
	read.W	rmw.D

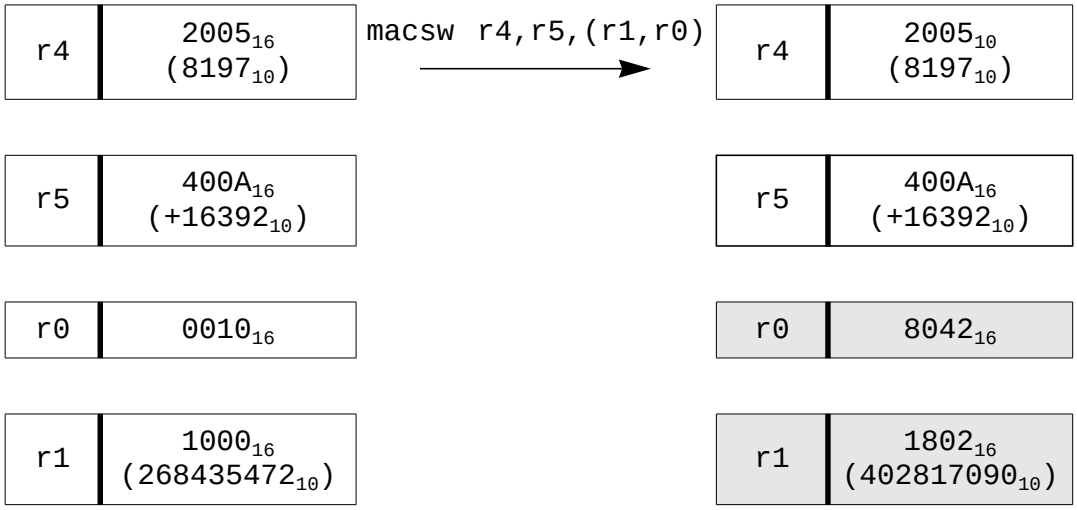
The **MACUW** instruction multiplies the ***src1*** operand by the ***src2*** operand and adds the result to the ***dest*** operand. The two src operands are interpreted as unsigned 16-bit integers. The result is viewed as unsigned 32-bit integer.

During the addition the result may overflow. In this case, the result is set (saturated) to the largest positive ($+2^{32}$-1) number.

Flag None

Trap None

Example Multiplies register R4 by register R5, and adds the result to the register pair (R1,R0).



MACQW	***src1***, ***src2***	***dest***
	reg.r, reg.r	reg.rp
	read.W	rmw.D

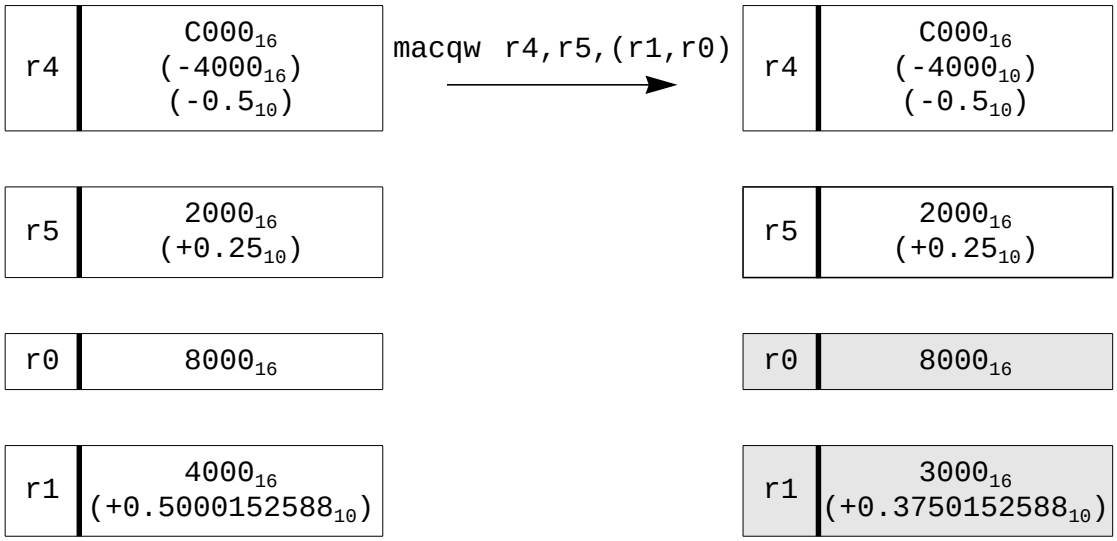
The **MACQW** instruction multiplies the ***src1*** operand by the ***src2*** operand and adds the result to the ***dest*** operand. The two src operands are interpreted as 16-bit signed fractional numbers (Q15 format). The result is viewed as a 32-bit fractional number.

During the multiplication or addition, the result may overflow or underflow. In either case, the result is set (saturated) to the largest positive (1) or largest negative (-1) number.

Flag None

Trap None

Example Multiplies register R4 by register R5, and adds the result to the register pair (R1,R0).



MOVi ***src***, ***dest***
reg.r/imm4/16reg.r
read.i write.i

MOVD ***src***, ***dest***
reg.rp/imm4/16/20/32reg.rp
read.D write.D

The **MOVi** instruction copies the ***src*** operand to the ***dest*** operand.

Flag

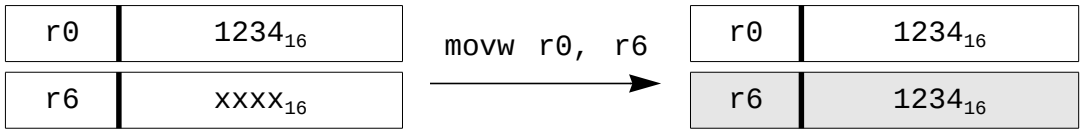
None

Trap

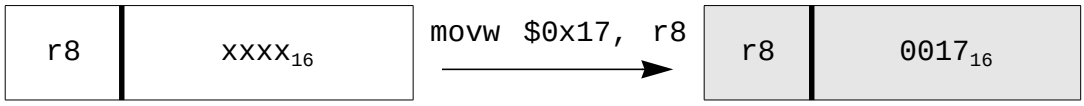
None

Examples

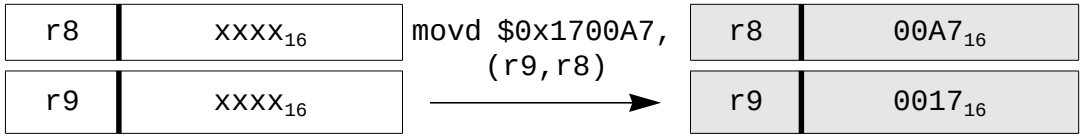
1. Copies the contents of register R0 to register R6.



2. Sets R8 to the value 17_{16}.



3. Sets register pair (R9, R8) to the value $\$0x1700A7_{16}$.



MOVXB, MOVXW

MOVXB	***src***,	***dest***
	reg.r	reg.r
	read.B	write.W
MOVXW	***src***,	***dest***
	reg.r	reg.rp
	read.W	write.D

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

Flag

None

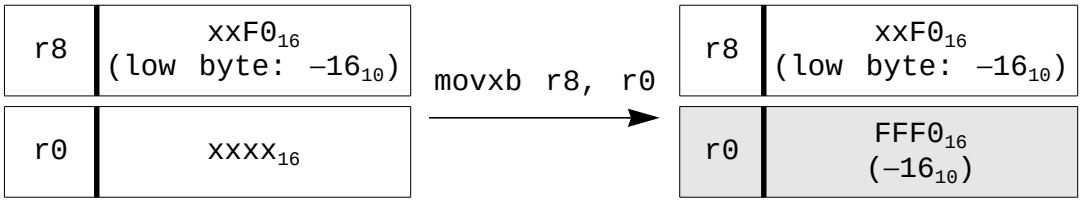
Trap

None

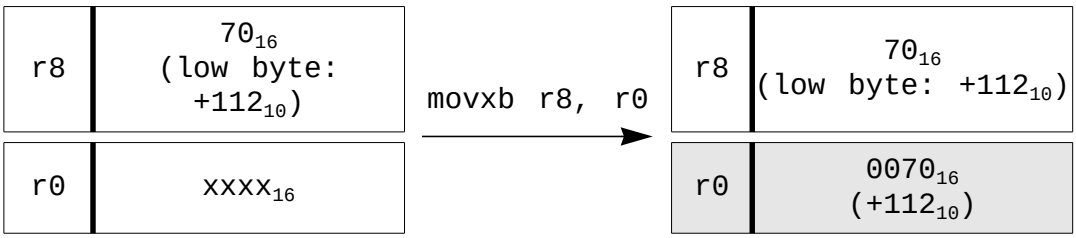
Examples

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 high-order bits of register R0.

1. Illustrates negative sign extension.



2. Illustrates positive sign extension.



MOVZB ***src***, d***est***
reg.r reg.r
read.B write.W

MOVZW ***src***, d***est***
reg.r reg.rp
read.W write.DThe **MOVZB** instruction converts the unsigned integer ***src*** operand to the unsigned word ***dest*** operand. The **MOVZW** instruction converts the unsigned integer ***src*** operand to the unsigned double-word ***dest*** operand. The high-order bits are filled with zeros.

Flag

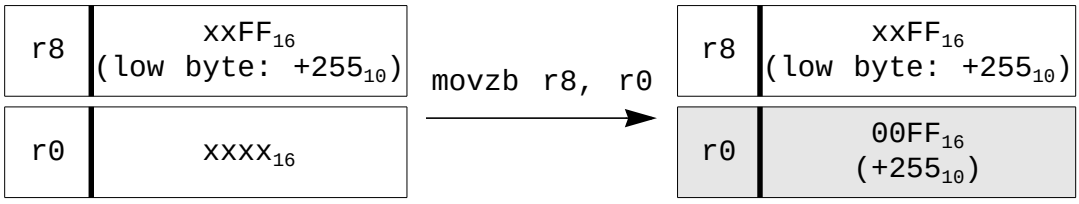
None

Trap

None

Example

Copies the low-order byte of register R8 to the low-order byte of register R0, and sets the high-order bits of register R0 to zero.



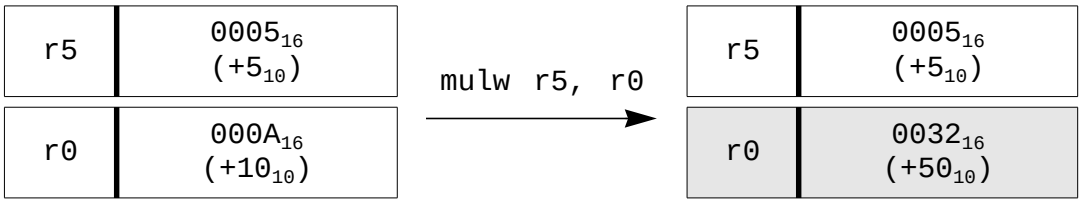
MULi ***src***, ***dest***
reg.r/imm4/16 reg.r
read.i rmw.i

The **MULi** instruction multiplies 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 completely in the ***dest*** operand, the high-order bits are truncated.

Flag None

Trap None

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



MULSB	***src***,	***dest***
	reg.r	reg.r
	read.B	rmw.W

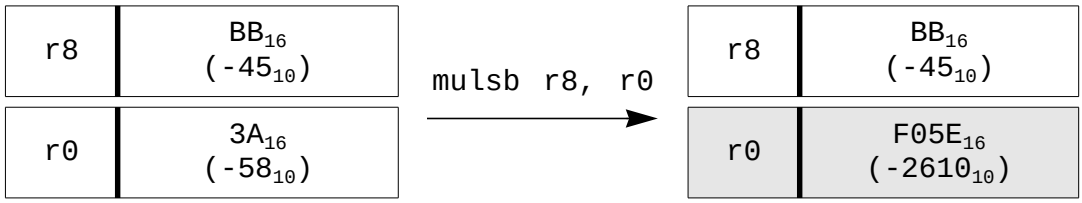
The **MULSB** instruction multiplies the 8-bit ***src*** operand by the 8-bit ***dest*** operand, and places the 16-bit result in the ***dest*** register.

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

Flag None

Trap None

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



MULSW	***src***,	***dest***
	reg	reg.rp
	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 ***dest*** operand.

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

Flag

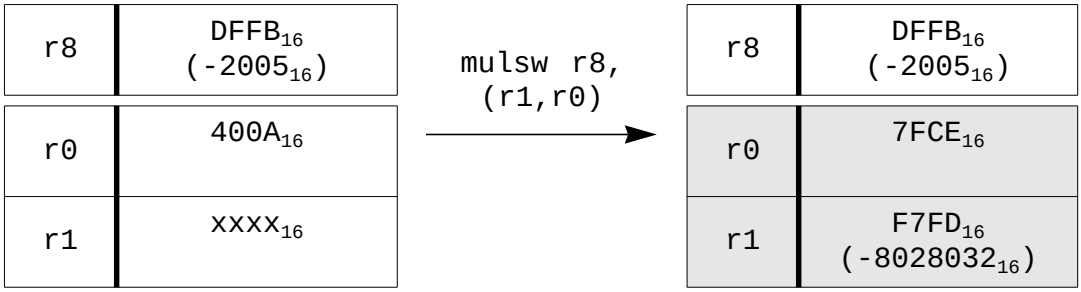
None

Trap

None

Example

Multiplies signed register R8 by R0, and places the result in registers (R1, R0).



MULUW	***src***,	***dest***
	reg	reg.rp
	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 ***dest*** operand.

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

Flag

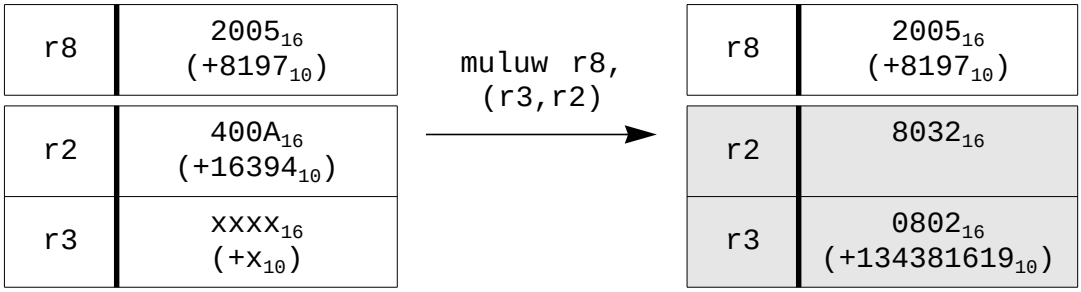
None

Trap

None

Example

Multiplies unsigned register R8 by R2, and places the result in registers (R3, R2).



NOP

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

Note: The NOP instruction is not natively supported by the core. It is mapped by the assembler to ADDUB $0x0, r0 (single word instruction).

Flag None

Trap None

Example

```
nop
```

ORi	***src***,	***dest***
	reg.r/imm4/16	reg.r
	read.i	rmw.i
ORD	***src***,	***dest***
	reg.rp/imm32	reg.rp
	read.D	rmw.D

The **ORi/ORD** instruction performs a bitwise logical OR operation on the ***src*** and ***dest*** operands, and places the result in the ***dest*** operand.

Flag

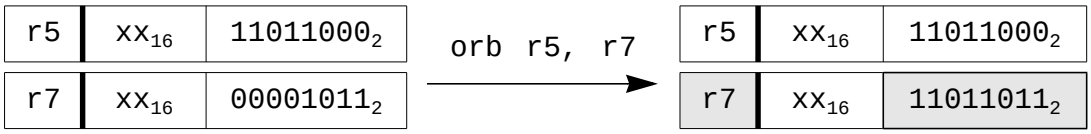
None

Trap

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	***count***,	***src***	***RA***
	imm3	reg.r	
	read	write.W	write.L
POPrt	***count***,	***src***	
	imm3,	reg.r	
	read,	write.W	
POPrt	***RA***		

write.LThe **POPrt** instruction can restore up to eight adjacent registers plus the ***RA*** from the program stack. The registers are defined by ***src*** register, with up to seven more adjacent registers (e.g., ***src***, ***src***+1, ***src***+2, ***src***+3, ***src+4***, ***src***+5, ***src***+6, ***src***+7) and the ***RA***. ***count*** reflects the total number of words to restore, excluding the ***RA***. The ***count*** operand is in the range 1 to 8.

src is loaded with the value residing at the lowest address (top of stack) and the ***RA*** register, if loaded, at the highest address. The stack pointer (**SP**) is adjusted (incremented) accordingly. Note that a double-word register is considered 2 words long. This instruction is not interruptible.

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**
none	no-return	--
RET	Return	**JUMP** RA

Depending on the format, the following registers are saved:

- In the three operand format, the instruction restores up to eight adjacent registers and the ***RA*** register.
- In the two operand format, the instruction restores up to eight adjacent registers.
- In the one operand format, the instruction restores only the RA register from the stack.

Note: The parameters to **POPrt** should not indicate a restore of registers R15 (**SP**) and beyond. This constrains the permitted count values as shown below:

RA is part of the instruction it refers to (ra) if CFG.SR = 0 and to (ra, era) otherwise

When CFG.SR = 1, parameters meeting the following condition are invalid:
if cnt + starting register number > 15

When CFG.SR = 0, parameters meeting the following conditions are invalid:

Starting register	Illegal cnt
0 -10	none
11	>7
12	>6
13	>4
14	>2
15	all

The **POPrt** 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.

Flag None

Trap None

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

PC	$00F034_{16}$		PC	$00F036_{16}$
SP	00001000_{16}		SP	00001006_{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 eight words from the stack starting with R8, pops RA, and executes a **JUMP RA**:

PC	$00F034_{16}$			PC	$866BD8_{16}$ (ra x 2}
SP	$000309f8_{16}$			SP	$0003100A_{16}$
r8	$XXXX_{16}$			r8	0000_{16}
r9	$XXXX_{16}$			r9	0001_{16}
r10	$XXXX_{16}$			r10	0002_{16}
r11	$XXXX_{16}$			r11	0003_{16}
r12	$XXXXXXXX_{16}$			r12	$107E2F50_{16}$
r13	$XXXXXXXX_{16}$			r13	$000335EC_{16}$
ra	$XXXXXXXX_{16}$			ra	$004335EC_{16}$
0309f8 (stack)	0000_{16}	popret $8,R8,RA	→	309f8 (stack)	$xxxx_{16}$ *
0309FA (stack)	0001_{16}			309FA (stack)	$xxxx_{16}$ *
0309FC (stack)	0002_{16}			309FC (stack)	$xxxx_{16}$ *
0309FE (stack)	0003_{16}			309FE (stack)	$xxxx_{16}$ *
031000 (stack)	$2F50_{16}$			031000 (stack)	$xxxx_{16}$ *
031002 (stack)	$107E_{16}$			031002 (stack)	$xxxx_{16}$ *
031004 (stack)	$35EC_{16}$			031004 (stack)	$xxxx_{16}$ *
031006 (stack)	0003_{16}			031006 (stack)	$xxxx_{16}$ *
031008 (stack)	$35EC_{16}$			031008 (stack)	$xxxx_{16}$ *
03100A (stack)	0043_{16}			03100A (stack)	$xxxx_{16}$ *

PUSH

PUSH	***count***,	***src,***	***RA***
	imm3	reg.r	
	read	read.W	read.L
PUSH	***count***,	***src***	
	imm3	reg.r	
	read	read.W	
PUSH	***RA***		

read.LThe **PUSH** instruction saves up to eight adjacent registers plus the ***RA*** on the program stack. The registers are defined by ***src*** register, with up to seven more adjacent registers (e.g., ***src***, ***src***+1, ***src***+2, ***src***+3, ***src+4***, ***src***+5, ***src***+6, ***src***+7) and the ***RA***. ***count*** reflects the total number of words to save, excluding the ***RA***. The ***count*** operand is in the range 1 to 8.

src is loaded with the value residing at the lowest address (top of stack) and the ***RA*** register, if saved, at the highest address. The stack pointer (**SP**) is adjusted (decremented) accordingly. Note that a double-word register is considered two words long. This instruction is not interruptible.

Register pairs stored within a single **PUSH** instruction comply with little-endian methodology.

Depending on the format, the following registers are saved:

- In the three operand format, the instruction saves up to eight adjacent registers and the ***RA*** register.
- In the two operand format, the instruction saves up to eight adjacent registers.
- In the one operand format, the instruction saves only the RA register from the stack.

Flag None

Trap None

RA is part of the instruction it refers to: (ra) if CFG.SR = 0, and (ra, era), otherwise.

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

SP	00001006_{16}
r5	$2F50_{16}$
r6	$107E_{16}$
r7	$35ED_{16}$
001000 (stack)	$XXXX_{16}$
001002 (stack)	$XXXX_{16}$
001004 (stack)	$XXXX_{16}$

push $3,r5

SP	00001000_{16}
r5	$2F50_{16}$
r6	$107E_{16}$
r7	$35ED_{16}$
001000 (stack)	$2F50_{16}$
001002 (stack)	$107E_{16}$
001004 (stack)	$35ED_{16}$

RETX

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

1. The instruction pops a 32-bit value from the interrupt stack, copying bits 22 - 0 to bits 23 - 1 of the PC.

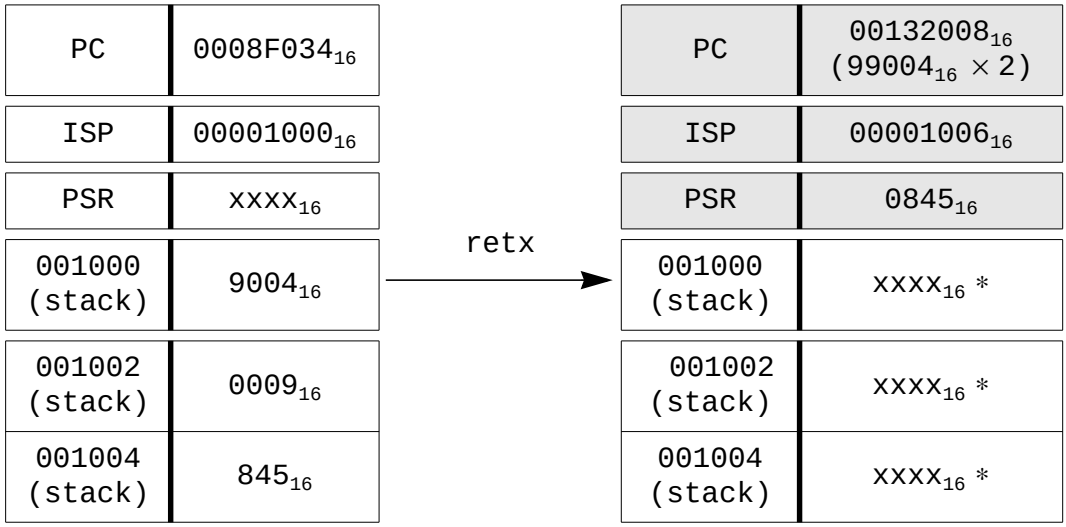
2. The instruction then pops a 16-bit value from the interrupt stack into the PSR register.

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.

Flag All PSR flag states are restored from the stack.

Trap None

Example Returns control from an interrupt service procedure.



SBITB	***position***,	***dest***
	imm3	abs/rel.r/rel.rp/idx
	read.i	read+write.i
SBITW	***position***,	***dest***
	imm4	abs/rel.r/rel.rp/idx
	read.i	read+write.i

The **SBITi** instruction loads the ***dest*** operand from memory, sets the bit position 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 **SBITB** is used, and 0 to +15 if **SBITW** is used; otherwise, the result is unpredictable.

As there is no native support for setting a bit in a double length core register, the ORD instruction should be used with SR=0.

See Table 5-4.

Flag

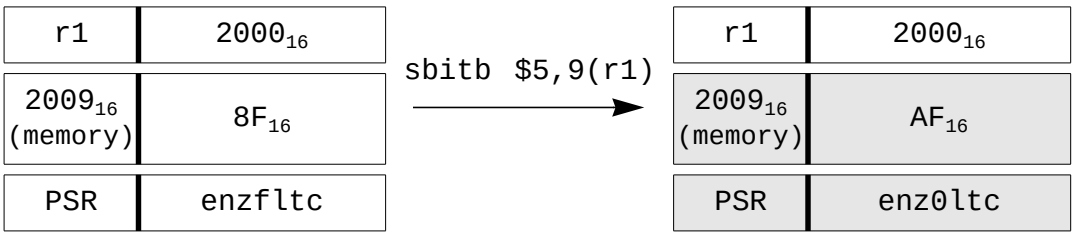
Before the specified bit is modified, its value is stored in PSR.F.

Trap

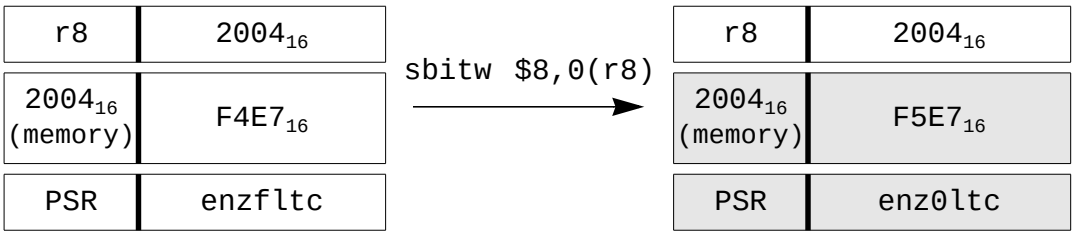
None

Examples

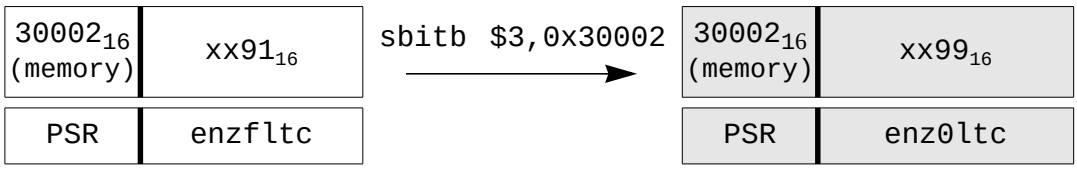
1. Sets bit in position 5 in a byte operand at address 9 (R1).



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}



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

Scond ***dest***
reg.r
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 the condition 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) is/are set as required by the specified ***cond***, the condition is true; otherwise, the condition is false. Table 5-9 describes the possible **cond** codes and the related PSR flag settings:

Table 5-9. *cond* Codes and Related PSR Flags

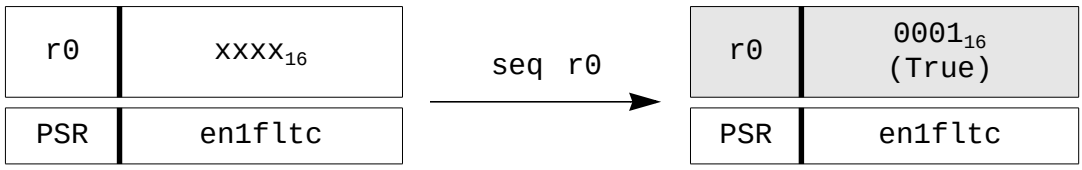
cond **Code**	**Condition**	**True State**
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal To	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag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

Flag None

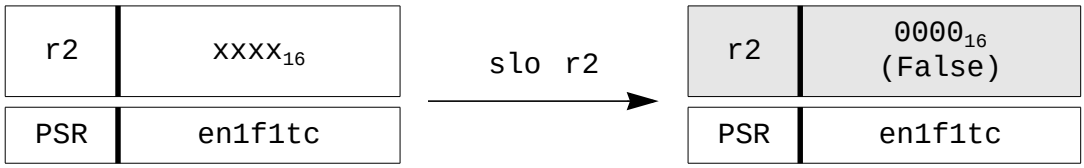
Trap None

Examples

1. Sets register R0 to 1 if the PSR.Z flag is 1, and to 0 if it is 0.



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



SPR	***src***,	***dest***
	procreg	reg.r
	read.W	write.W
SPRD	***src***,	***dest***
	procregdreg.rp	
	read.D	write.D

The **SPR/SPRD** instruction copies the processor register specified by the ***src*** operand to the ***dest*** operand.

On a SPRD of a 16 bit register, the upper 16 bits of the dest are loaded with 0.

The processor registers in Table 5-10 may be stored:

Table 5-10. Storable Processor Registers

Register	SPR	SPRD
Processor Status Register	**PSR**	**PSR**
Configuration Register	**CFG**	**CFG**
Interrupt Base Register		**INTBASE**
Interrupt Base Low Register	**INTBASEL**	
Interrupt Base High Register	**INTBASEH**	
Interrupt Stack Pointer Register		**ISP**
Interrupt Stack Pointer Low Register	**ISPL**	
Interrupt Stack Pointer High Register	**ISPH**	
User Stack Pointer Register		**USP**
User Stack Pointer Low Register	**USPL**	
User Stack Pointer High Register	**USPH**	
Debug Status Register	**DSR**	**DSR**
Debug Condition Register		**DCR**
Debug Condition Low Register	**DCRL**	
Debug Condition High Register	**DCRH**	
Compare Address 0 Register		**CAR0**
Compare Address 0 Register Low	**CAR0L**	
Compare Address 0 Register High	**CAR0H**	

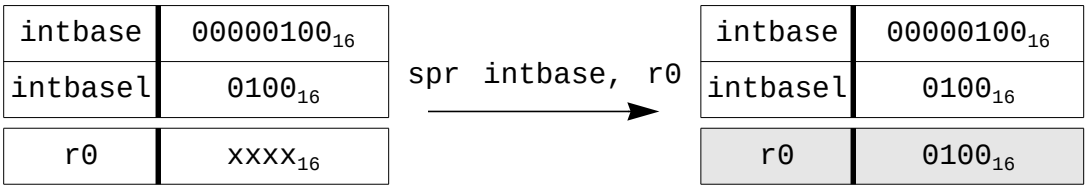
Register	SPR	SPRD
Compare Address 1 Register		**CAR1**
Compare Address 1 Register Low	**CAR1L**	
Compare Address 1 Register High	**CAR1H**	

Refer to “REGISTER SET” on page 3-1 and to “INSTRUCTION SET” on page 6-1 for more information on these registers.

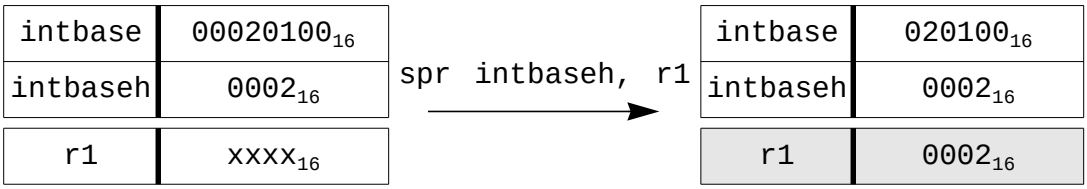
Flag None

Trap None

Examples 1. Copies the INTBASEL register to register R0.



2. Copies the INTBASEH register to register R1.



STORi	***src***,	***dest***
	reg.r	abs/rel.r/rel.rp/idx
	read.i	write.i
STORD	***src***,	***dest***
	reg.rp	abs/rel.r/rel.rp/idx
	read.D	write.D
STORi	***src***,	***dest***
	imm4	abs/rel.r/rel.rp/idx
	read.i	write.i

The **STORi** instruction stores the ***src*** operand in the ***dest*** memory operand.

Table 5-7 describes the addressing options for the first and second formats of the instruction.

The third format of the instructions allows storing an unsigned 4-bit immediate operand (in the range of 0 to 15) into the ***dest*** memory operand. The addressing options for this format are described in Table 5-4.

Flag

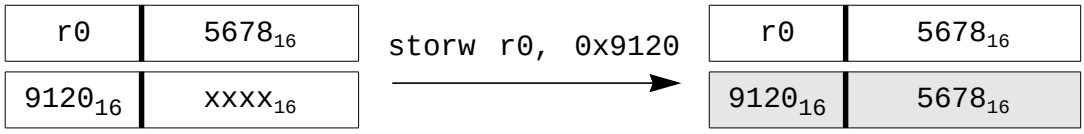
None

Trap

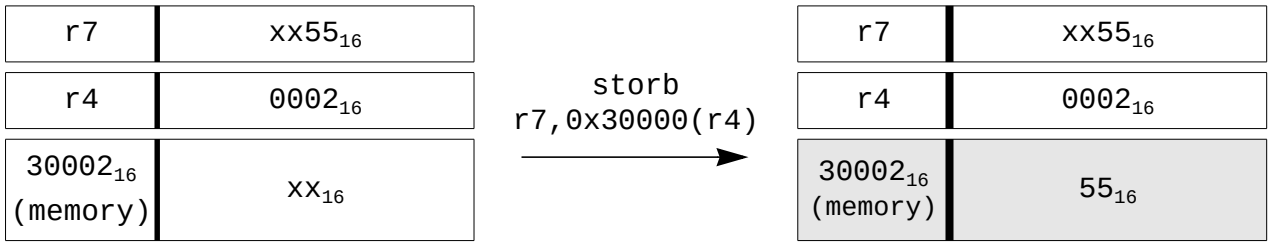
None

Examples

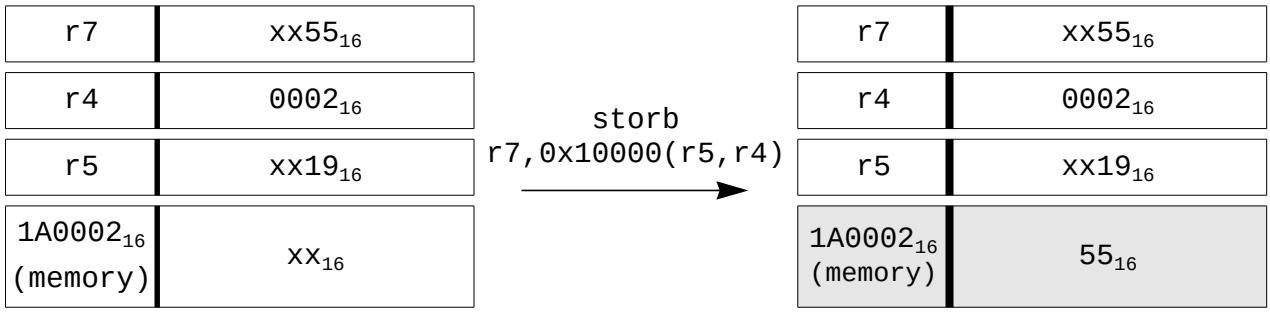
1. Copies the contents of register R0 to the word at address 9120_{16}.



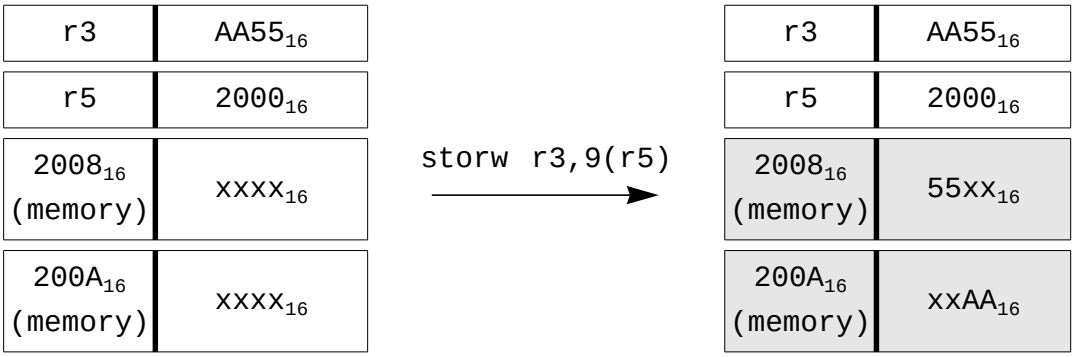
2. Stores the low-order byte of R7 at address 30002_{16}. The address is formed by adding 30000_{16} to the value in R4.



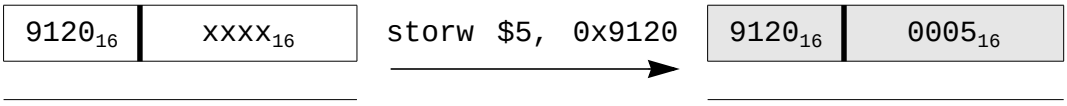
3. Stores the low-order byte of R7 at address $1A0002_{16}$. The address is formed by adding 10000_{16} to the value in R4, concatenated with the value in R5.



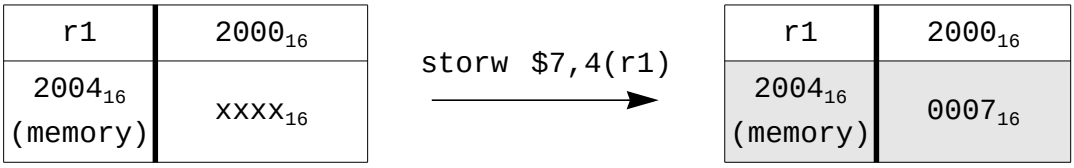
4. Copies the contents of register R3 to the non-aligned word at address 9 (R5).



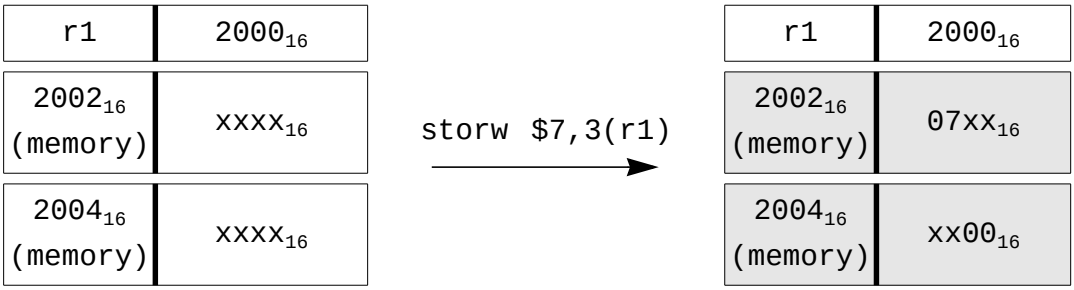
5. Stores the immediate value of 0x5 to the word at address 9120_{16}.



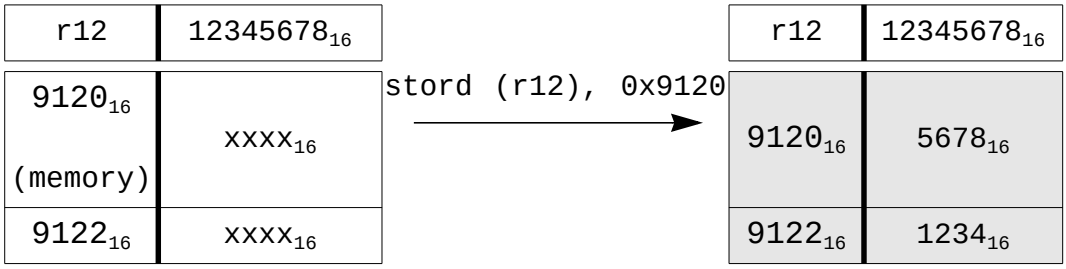
6. Stores the immediate value of 0x7 to the word at address 4 (R1).



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



8. Copies the contents of register R12 to the double-word at address 9120_{16}



STORM ***count***
imm3
read,

STORMP ***count***
imm3
read,

The **STORM** and **STORMP** instructions store adjacent registers to memory. ***count*** reflects the total number of words to be stored where ***count*** is in the range from 1 to 8. The instructions always operate on a fixed set of registers.

For **STORM**:

- R1 contains the target address of the first word in memory;
- R2 is stored into the lowest address word;
- R3 through R5 and then R8 through R11 are stored into 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. The address does not wrap around i.e. if R1 points to the end of the 64k addressable range STORM will overflow to addresses 0x010000 and following.

For **STORMP**:

- (R7,R6) contains the target address of the first word in memory;
- R2 is stored into the lowest address word;
- R3 through R5 and then R8 through R11 are stored into the next ***count***-1 consecutive addresses.

(R7,R6) 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 interruptible.

Flag None

Trap None

Example Stores three registers into memory.

r1	1000_{16}
r2	$2F50_{16}$
r3	$107E_{16}$
r4	$35EC_{16}$
001000 (memory)	$XXXX_{16}$
001002 (memory)	$XXXX_{16}$
001004 (memory)	$XXXX_{16}$

storm $3 →

r1	1006_{16}
r2	$2F50_{16}$
r3	$107E_{16}$
r4	$35EC_{16}$
001000 (memory)	$2F50_{16}$
001002 (memory)	$107E_{16}$
001004 (memory)	$35EC_{16}$

SUBi	***src***,	***dest***
	reg.r/imm4/16	reg.r
	read.i	rmw.i
SUBD	***src***,	***dest***
	reg.rp/imm32	reg.rp
	read.D	rmw.D

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

Flag

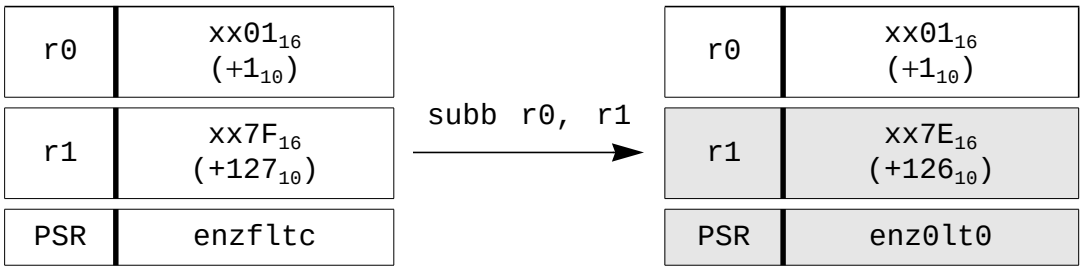
During execution of an **SUBi or SUBD** instruction, PSR.C is set to 1 if a borrow occurs, and cleared to 0 if no borrow occurs. PSR.F is set to 1 if an overflow occurs, and cleared to 0 if there is no overflow.

Trap

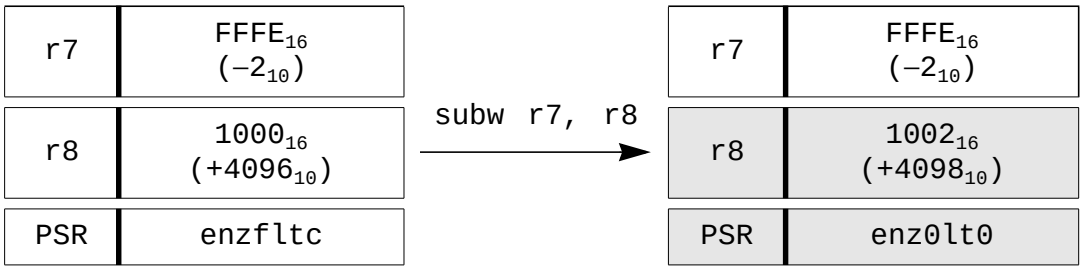
None

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.



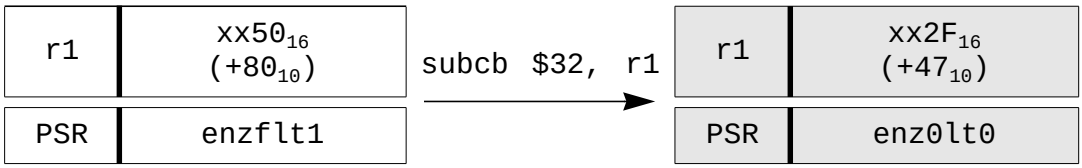
SUBCi ***src***, ***dest***
reg.r/imm4/16reg.r
read.i rmw.i

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

Flag 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.

Trap 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	***offset,***	***src***
	reg.r/imm4	reg.r
	read.W	read.W
TBITi	***offset***,	***src***
	imm4	abs/rel.r/rel.rp/idx
	read.i	read.i

The **TBIT** instruction copies the bit located in register or memory location ***src*** at the bit position specified by ***offset***, to the PSR.F flag. The direct memory format of the instruction supports byte and word operations (**TBITB**, **TBITW**), while the register-sourced format supports only word operations. The ***offset*** value must be in the range of 0 through 15 for a word operand, and in the range of 0 through 7 for a byte operand.

As there is no native support for testing a bit in a double-length core register with SR=0, an ANDD/CMPD instruction sequence should be used.

See Table 5-4.

Flag

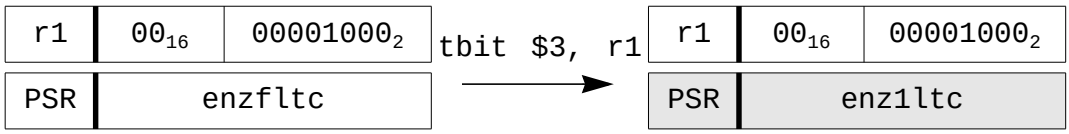
PSR.F is set to the value of the specified bit.

Trap

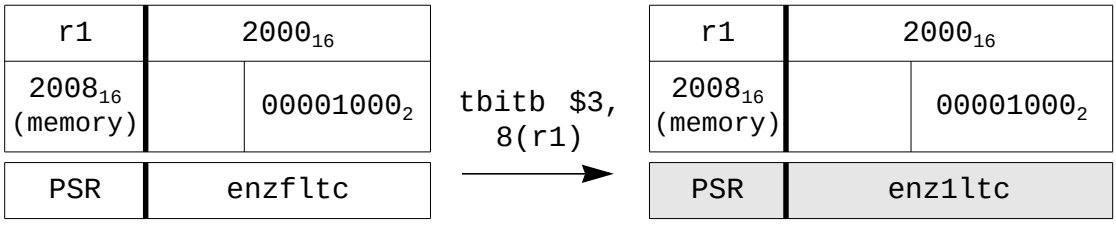
None

Examples

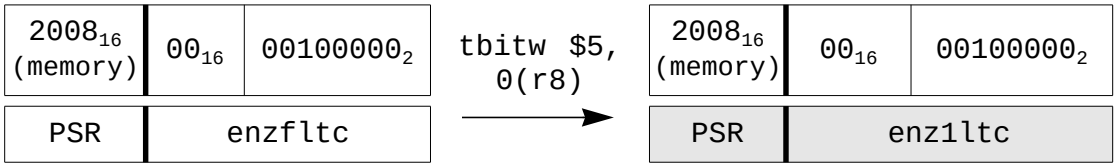
1. Copies bit in position 3, in register R1 to the PSR.F flag.



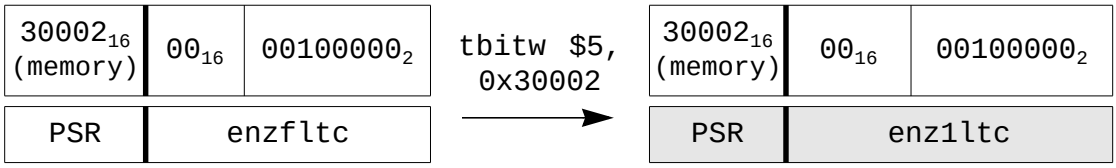
2. Copies bit in position 3, in memory location 8 (R1) to the PSR.F flag.



3. Copies bit in position 5, in memory location 0 (R8) to the PSR.F flag.



4. Copies bit in position 5, in memory location 30002_{16} to the PSR.F flag.



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.

Flag None

Trap None

Example

```
wait
```

XORi	***src***,	***dest***
	reg.r/imm4/16	reg.r
	read.i	rmw.i
XORD	***src***,	***dest***
	reg.rp/imm32	reg.rp
	read.D	rmw.D

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

Flag

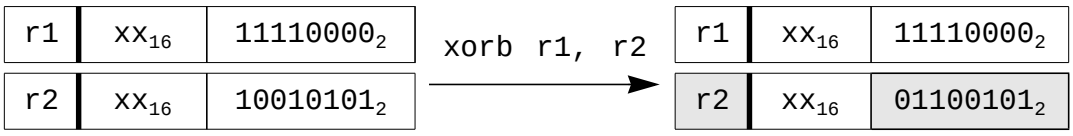
None

Trap

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.



This page is intentionally blank.

Appendix A
INSTRUCTION EXECUTION TIMING

This appendix describes the factors which affect instruction execution timing in the CR16C. A CR16C-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. For example, at 50 MHz operation, there are 50 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 it.

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

Program Execution Time - The number of elapsed clock cycles from the time the first instruction begins until the last instruction leaves 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 CR16C performs one instruction per clock cycle. At 50 MHz, this translates to 50 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.

During an instruction fetch when 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.

Calculating program 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 CR16C passes through three stages in the pipeline:

- Instruction Fetch (IF)
- Instruction Decoding (ID)
- Instruction Execution (EX)

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

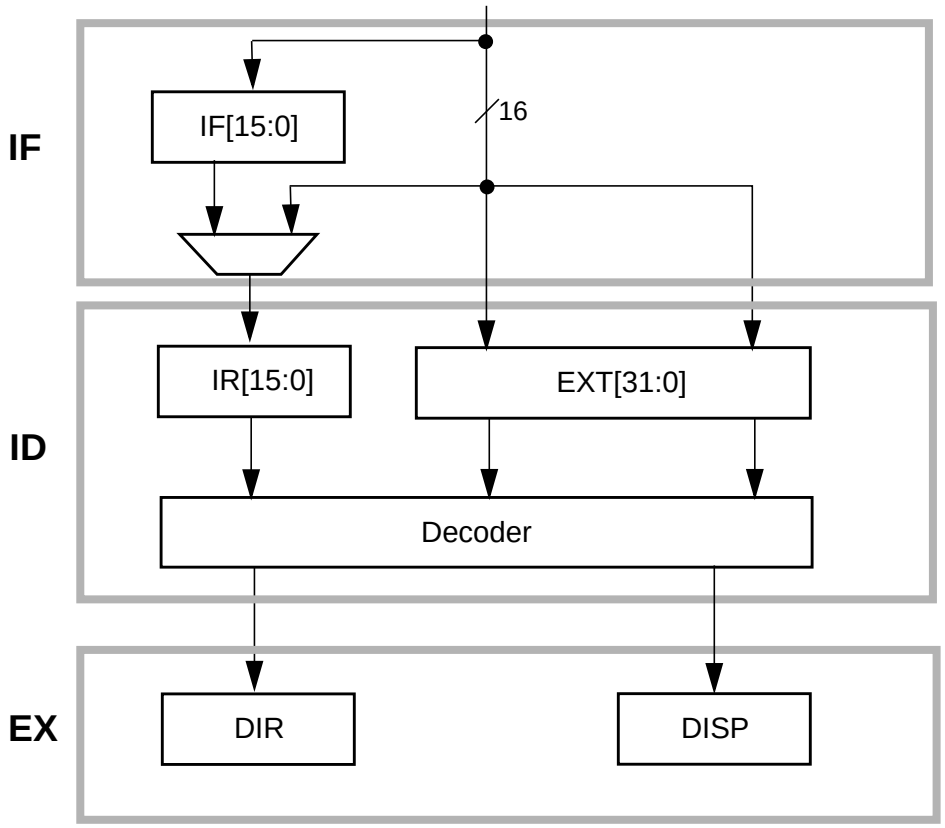


Figure A-1. Instruction Flow through the Pipeline

The IF stage

The CR16C fetches the first word of an instruction directly into the Instruction Register (IR) if the IR is idle. If it is not idle and a fetch is executed, the next instruction is stored in the next available Fetch register until the IR is free again. The consecutive words of a two- or three-word instruction are directly fetched into the upper or lower half of the 32-bit Extension register (EXT). Initiating the fetch cycle is controlled by the ID stage, such that at the end of the fetch cycle, the IR and, depending on the length of the instruction, EXT hold a valid instruction.

The ID stage

The Instruction Decode (ID) stage uses two registers, IR and EXT to decode the current instruction. In combination, these registers hold the instruction's opcode and the displacement, absolute address or immediate value. The ID stage is fully loaded with a complete instruction before decode begins. At the end of the decode stage the execution control registers are loaded.

The EX stage

Instructions are executed in the Execution (EX) stage. The currently executing instruction is held in the Decoded Instruction Register (DIR) and the corresponding displacement or constant is held in the DISP register until execution ends.

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 branch or jump instruction:

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

Source operands are read in the ID stage and results are written only in the EX stage. The instruction latency, when there are no delays, is usually three clock cycles (from fetch until execution end).

Instruction completion

The instruction execution state machine signals End of Instruction (EOI) when it starts the last cycle of an instruction. For non-data-transfer related instructions, this indicates that the current instruction terminates after this cycle. Data transfer instruction completion may be delayed, if the data transfer is not completed, until the RDY signal is asserted for the last transferred data portion.

Rolling the pipeline

At best, the CR16C pipeline proceeds to the next stage once every clock. The core fetches a new instruction, or continues to fetch the second word of a previously fetched instruction, whenever the IF stage is going to be empty the following cycle. The conditions for issuing a new fetch cycle can be one of the following:

1. The pipeline has been flushed (e.g., during a Branch instruction) and therefore all of its stages are empty.

2. The currently executing instruction ends during this cycle.

3. Either the Instruction Register (IR) is free and IF contains the first word of an instruction, or the Extension Register (EXT) is free and IF contains the second word of an instruction. (e.g., when the fetch phase takes more than a clock due to bus latencies).

In such cases, the core performs the following operations:

- It schedules the currently decoded instruction, if one exists, to be transferred to the EX stage.

- It schedules the transfer of the previously fetched instruction (first or second word), residing in the IF register into the decode stage (ID or EX respectively).

- It schedules a new instruction fetch cycle that results in loading a new instruction word into the IF register.

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

Decoding of an instruction, i.e., the ID stage, is delayed until all of its words have been fetched, i.e., are completely in the IF stage. Depending on the length of the instruction up to three bus cycles are required to fetch this instruction.

For example, it takes two bus cycles to fetch a double-word instruction from memory. If no instruction is in the ID stage when the example 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 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 state is needed while the memory from which instructions are fetched is accessed, the overall delay for a branch is three clock cycles for a single word target instruction plus potential additional cycles if the target instruction is longer.

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 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.

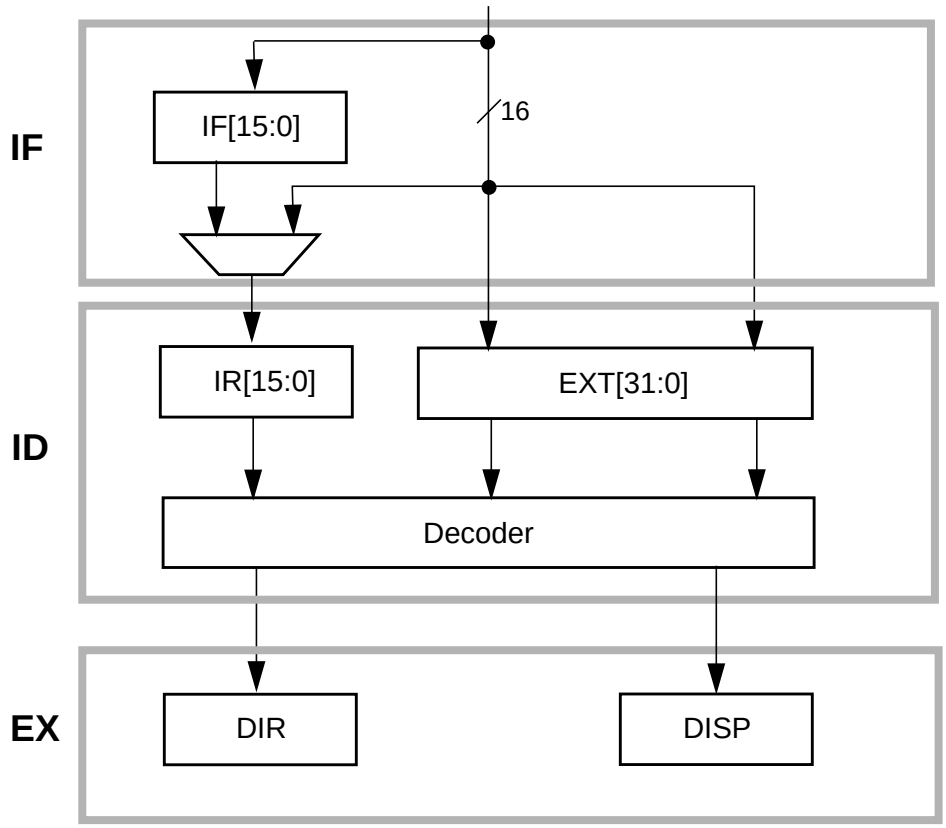
Store delays

A store instruction is executed in two 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.

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 Section 4.3.2 on page 4-26 for more information on serializing instructions.



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 Section A.3.

Arithmetic instructions

Table A-1. Execution Times for Arithmetic Instructions

Instruction	Clock Cycles in EX Stage
ADDi, ADDCi, ADDUi, ANDi, ASHUi, CMPi, LSHi, MOVi, MOVX, MOVZ, ORi, Scond, SUBCi, SUBi, NOP, TBIT and XORi.	1
ASHUD, LSHD,CMDD,ADDD	1
MOVXW, MOVZW	1
MOVD	1
MULB	1
MULW	4

Table A-1. Execution Times for Arithmetic Instructions

Instruction	Clock Cycles in EX Stage
MULSB	2
MULSW	4
MULUW	4
MACSW, MACUW	6
MACQW	7

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 that require wait states are fetched from memory, e.g., off-chip memory.

In this case, additional clock cycles may be added to the number shown in the table. depending 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/Store Instructions

Instruction	Clock Cycles in EX Stage (Zero Wait States)	Memory Alignment	Bus Accesses in EX Stage (Zero Wait States)
LOADB	2	aligned	1
LOADB	3	non-aligned	1
LOADW	2	aligned	1
LOADW	4	non-aligned	2
LOADD	3	aligned	2
LOADD	5	non-aligned	3
STORB	2	aligned	1
STORB	2	non-aligned	1
STORW	2	aligned	1
STORW	3	non-aligned	2
STORD	3	aligned	2
STORD	4	non-aligned	3

PUSH, POP/POPRET, 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 instructions 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 that require wait states are fetched from memory, e.g., off-chip memory.

In this case, additional clock cycles may be added to the number shown in the table, depending on the speed at which memory can be accessed.

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

Table A-3. Execution Times for PUSH, POP/POPRET, LOADM and STORM

Instruction	N[a]	Clock Cycles in EX Stage (Zero Wait States)	Memory Address / Stack Alignment	Bus Accesses in EX Stage
PUSH[N]/ STORM[N]	(1-8)	N+1	Aligned	N
		N+2	Not word aligned	N+1
PUSH[N}, RA	(1-8) + 2	N+1	Aligned	N
		N+3	Not word aligned	N+2
LOADM[N] / POP[N]	(1-8)	N+2	Aligned	N
	2,4,6,8	2N+2	Not word aligned	1.5N
	1,3,5,7	2N+3		0.5(3N+1)
POP[N], RA	(1-10)	N+2	Aligned	N
	2,4,6,8,10	2N+2	Not word aligned	1.5N
	1,3,5,7,9	2N+3		0.5(3N+1)
+RET (for POP)		+1[b]	always	-

a. N = Number of registers to load or store

b. Note that **POPRET**, like the **BR** and **JUMP** derivatives, flushes the pipeline. Therefore, when calculating total throughput, the appropriate branch delay should be accounted for.

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 that require wait states are fetched from memory, e.g., off-chip memory.

In this case, additional clock cycles may be added to the number shown in the table, depending on the speed at which memory can be accessed.

When zero wait states is used, the number of additional cycles that is needed to execute the instructions depends 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)	Bus Accesses in EX Stage
SBITi/CBITi	4	2
TBITi	3	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	Clock Cycles in EX Stage	Pipeline Flush Condition
LPR	2	Always
SPR	1	Never
JUMP, BAL, JAL, BR	1	Always
Bcond	1	When condition is true
Compare & Branch	2	When condition is true
RETX	5	Always
WAIT, EIWAIT	1	Always, by interrupt
EI, DI	1	Never

Interrupts and traps

The interrupt or trap latency of the CR16C 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 CR16 instruction (**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.

Table A-6. Execution Times for Interrupts and Traps

Interrupt or Trap	Clock Cycles in EX Stage	
	CFG.ED=0 (16-bit dispatch table)	CFG.ED=1 (32-bit dispatch table)
INT	12	13
NMI and ISE[a]	11	12
TRC[a]	9	10
DBG[a]	9	10
SVC, DVZ, FLG, BPT[a] and UND	9	10

a. For ISE, TRC, DBG and BPT, if AISE, ATRC, ADBG and ABPT bits in the CFG registers are set (respectively), then the execution time is that of ISE/NMI for Large Model the 32-bit dispatch table (12 cycles). See Section 4.1.5 on page 4-14 for more details.

This page is intentionally blank.

Appendix B
INSTRUCTION SET ENCODING

This appendix describes instruction set encoding. Most instructions are encoded according to the basic instruction format. The various instruction formats are detailed and then the instructions for each class of instructions are listed, followed by a summary of addressing methodology.

B.1 INTRODUCTION

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

The most frequently used instructions are encoded into one word. These instructions have zero to three operands and use the basic formats, shown in Figure B-1.

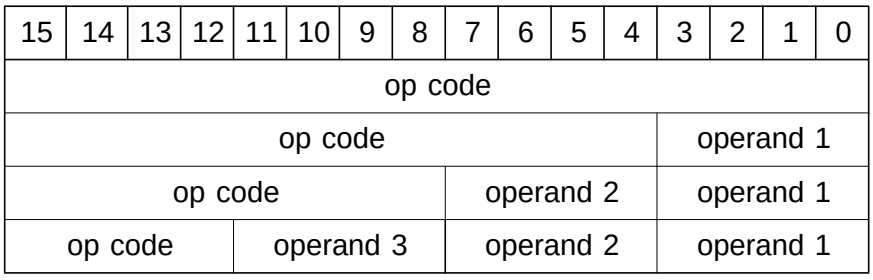


Figure B-1. Basic Instruction Structure: First Word

The operands are typically encoded in 4-bit fields, starting from the least significant bit of the word. The opcode typically uses all available bits, starting from the most significant bits of the word.

Instructions using 16-bit displacements or 20-bit labels require an additional word in the instruction. In these cases, the second word typically contains bits 15 to 0 of the displacement or label, as shown in Figure B-2 below.

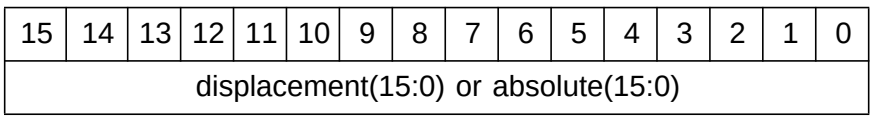


Figure B-2. Basic Instruction Structure: Second Word

Certain less frequently used operations using a 20/24-bit displacement/label or a 32-bit immediate use three-word encoding. In these cases, the first word is used as an escape to indicate that the following two words contain the instruction. These two words then use the two-word instruction format with an opcode and operands in the first word, and a displacement/label or immediate in the second word.

B.2 INSTRUCTION FORMATS

Most instructions use one of the basic formats described above; the rest use slight variations thereof. Opcode formats are described in Table B-1 below. These formats are referred to by their number in the instruction descriptions.

Table B-1. CR16C Opcode Formats

				16 word1 at addn 0		16 word2 at addn+2 0				16 word3 at addn+4 0	
fmt #	**format name**	**# wrds**	**oc size**	**byte 1**	**byte 0**	**byte 3**		**byte 2**		**byte 5**	**byte 4**
1	escape2	2	16	opcode16		p4_4	p3_4	p2_4	p1_4		
2	escape3_20	3	16+4[a]	opcode16		p4_4	p3_20	p2_4	p1_4	p3_20	
3	escape3_24	3	16+4[a]	opcode16		p4_4	p1_24(19:16)	p2_4	p1_24 (23:20)	p1_24	
3a	escape3a_24	3	16+4[a]	opcode16		p4_4	p1_24(19:16)	p2_4	p1_24 (23:20)	p1_24(15:1)	24
4	param0	1	16	opcode16							
5	param0_24	2	8	opcode8	p1_24(23:16)	p1_24(15:1)				24	
6	param3	1	13	opcode13	p1_3						
7	param3_20	2	9	opcode9	p2_3 p1_20	p1_20					
8	param31_20	2	8	opcode8	p3 p2_3 p1_20	p1_20					
9	param34	1	9	opcode9	p2_3 p1_4						
10	param34_16	2	9	opcode9	p2_3 p1_4	p3_16					
11	param4	1	12	opcode12	p1_4						
12	param4_20	2	8	opcode8	p2_4 p1_20	p1_20					
13	param41_20	2	7	opcode7	p3 p2_4 p1_20	p1_20					
14	param431	1	8	opcode8	p3 p2_3 p1_4						
15	param44	1	8	opcode8	p2_4 p1_4						
16	param44_16	2	8	opcode8	p2_4 p1_4	p2_16					
17	param44_14	2	10	opcode10	p3 5:4 p1_4	p3_14 (7,6,13-8)		p2_4	p3_14 (3:0)		

				16 word1 at addn 0				16 word2 at addn+2 0		16 word3 at addn+4 0	
fmt #	format name	# wrds	oc size	byte 1		byte 0		byte 3	byte 2	byte 5	byte 4
18	param444	1	4	op-code4	p3_4	p2_4	p1_4				
19	param444_16	2	4	op-code4	p3_4	p2_4	p1_4	p4_16			
20	param54	1	7	opcode7		p2_5	p1_4				
21	param84	1	4	op-code4	p1_8(8:5)	p2_4	p1_8(4:1)				
22	param84_16	2	4	op-code4	p1_8(8:5)	p2_4	p1_8(4:1)	p3_16(15:1)	16		
23	param4_32	3	12	opcode12			p1_4	p2_32			

a. The first word of the instruction is an escape code, and pt_4 is the expansion opcode.

B.2.1 Field Definitions for CR16C Encoding

The following tables detail the parameter types and modes used in the instruction fields.

Table B-2. Parameter Types and Modes

Parameter Types	Description	Parameter Modes
dest	destination	reg, rp, rrp, rs, disp, disp2, (rp), (reg), (prp), (rrp), abs, pr
src	source	reg, rp, rrp, rs, disp, disp2, (rp), (reg), (prp), (rrp), abs, imm, pr
link	link pointer	rp
ope	opcode extension	imm
vect	exception vector	imm
ci	co-processor index	imm
cinst	coprocessor instruction	imm
cond	condition code	imm
count		imm, reg
pos	position	imm, reg
res	reserved	- set to 0 -

Table B-3. Parameter Descriptions

Parameter	Value	Meaning	Description
rs[a]	0	R12 selected	index register select
	1	R13 selected	
RA	0	RA not used	RA select for stack operations

Parameter	Value	Meaning	Description
	1	RA used	
reg(4)	0	R0	register number
	1	R1	
	2	R2	
	3	R3	
	4	R4	
	5	R5	
	6	R6	
	7	R7	
	8	R8	
	9	R9	
	10	R10	
	11	R11	
	12	R12_L	least significant 16 bits of R12
	13	R13_L	least significant 16 bits of R13
	14	RA_L	least significant 16 bits of RA
	15	SP_L	least significant 16 bits of (U)SP (if PSR.U set: use USP_L else SP_L)
rp(4)	0	R1,R0	register pair
	1	R2,R1	
	2	R3,R2	
	3	R4,R3	
	4	R5,R4	
	5	R6,R5	
	6	R7,R6	
	7	R8,R7	
	8	R9,R8	
	9	R10,R9	
	10	R11,R10	
	11	R12_L,R11	
	12	R12	if CFG.SR set: use R13_L, R12_L
	13	R13	if CFG.SR set: use RA_L, R13_L
	14	RA	if CFG.SR set: use (U)SP_L, RA_L (if PSR.U set: use USP_L else SP_L)
	15	SP	if CFG.SR set: use (U)SP_H, (U)SP_L (if PSR.U set: use USP else SP)
rrp(4)[a]	0	R12 + R1,R0	reduced register pair
	1	R12 + R3,R2	
	2	R12 + R5,R4	
	3	R12 + R7,R6	

Parameter	Value	Meaning	Description
	4	R12 + R9,R8	
	5	R12 + R11,R10	
	6	R12 + R4,R3	
	7	R12 + R6,R5	
	8	R13 + R1,R0	
	9	R13 + R3,R2	
	10	R13 + R5,R4	
	11	R13 + R7,R6	
	12	R13 + R9,R8	
	13	R13 + R11,R10	
	14	R13 + R4,R3	
	15	R13 + R6,R5	
prp(4)		if CFG.SR set: prp is reg else prp is rrp[a]	
prd(4)	0	DBS	processor registers for LPRD/SPRD
	1	DSR	
	2	DCR	
	3	reserved	
	4	CAR0	
	5	reserved	
	6	CAR1	
	7	reserved	
	8	CFG	
	9	PSR	
	10	INTBASE	
	11	reserved	
	12	ISP	
	13	reserved	
	14	USP	
	15	reserved	
pr(4)	0	DBS	processor registers for LPR/SPR
	1	DSR	
	2	DCRL	
	3	DCRH	
	4	CAR0L	
	5	CAR0H	
	6	CAR1L	
	7	CAR1H	
	8	CFG	
	9	PSR	

Parameter	Value	Meaning	Description
	10	INTBASEL	
	11	INTBASEH	
	12	ISPL	
	13	ISPH	
	14	USPL	
	15	USPH	
vect	0		exception vector
	1		
	2		
	3		
	4		
	5	SVC	
	6	DVZ	
	7	FLG	
	8	BPT	
	9	TRC	
	10	UND	
	11		
	12	IAD	
	13		
	14	DBG	
	15	ISE	
cond(4)	0	eq	compare conditions
	1	ne	
	2	cs	
	3	cc	
	4	hi	
	5	ls	
	6	gt	
	7	le	
	8	fs	
	9	fc	
	10	lo	
	11	hs	
	12	lt	
	13	ge	
	14	always	
	15	uc	unconditional for jcond also set U flag (JUSR)

Parameter	Value	Meaning	Description
pos(3)	0	bit0	bit position
	1	bit1	
	2	bit2	
	3	bit3	
	4	bit4	
	5	bit5	
	6	bit6	
	7	bit7	
pos(4)	0	bit0	bit position
	1	bit1	
	2	bit2	
	3	bit3	
	4	bit4	
	5	bit5	
	6	bit6	
	7	bit7	
	8	bit8	for byte ops bit 0
	9	bit9	for byte ops bit 1
	10	bit10	for byte ops bit 2
	11	bit11	for byte ops bit 3
	12	bit12	for byte ops bit 4
	13	bit13	for byte ops bit 5
	14	bit14	for byte ops bit 6
	15	bit15	for byte ops bit 7
ope(4)	0-15	Opcode Extension	used as an extension to an opcode for codes encoded in three words.
(rp)		data at address rp	
(rrp[a])		data at address rrp	
(reg)		data at address reg	
rp*2		jump to address of rp shifted once left	
abs		absolute address - used as explicit address pointer	
rel		index register relative - the contents of an index register is added to the address	
imm4	0-15	For Store Immediate	
imm4	0-8,10, 12-15	In arithmetic operations, when parameter type is source, used as immediate values	
	9	interpreted as -1	
	11	used as escape code to 16 bit immediate format	

Parameter	Value	Meaning	Description
imm16		signed[b] immediate value	
imm20		unsigned immediate value	
imm32		signed[b] immediate value	
disp		immediate value used as displacement- added to other source or destination parameters	
disp*2		displacement value shifted once left - multiplied by 2	
disp*2+		displacement value add 1 and shift once left for beq0/bne0 disp4 commands (+2 to +32)	
disp4	0-13	treated as a positive displacement for store/load word/double-word operations shift once left	
	14	escape used to denote a prp register mode operation with disp0	
	15	used as escape code to displacement 16 format	
disp8	not 0x80	disp of 8 bits is always signed and shifted by 1 (mult by 2)	
	0x80	treated as escape to displacement16 format (format #22)	
disp(n) where n>8		for branches, disp of 8 bits or more is signed and shifted by 1 (mult by 2)	
		for loads, stores and bitops, disp(n) is always unsigned and not shifted!	
-disp20	-2^{20} to -1	for backwards compatibility with CR16B for load/stor instructions	

a. When CFG.SR =1, the index addressing mode is not supported. All instructions using "rrp" or "rs" cause undefined behavior.

b. The assembler must allow unsigned values for logic operations such as AND, OR, XOR using this encoding.

B.3 CR16C INSTRUCTION SET SUMMARY

For each class of instructions listed in Tables B-4 through B-14, the instructions are detailed with:

- the 16-bit binary opcode of the first word of the instruction
- Parameter values (number bits, type- pt?, mode-pm?)
- the opcode format number (fmt#)
- the opcode length (ln).

An “x” in the opcode value denotes bit positions reserved for parameters, as determined by the format.

Due to the structure of the opcode assignment, items such as the format type, byte/word command, arithmetic/logical operations, and field placement are readily available from a limited decode.

Table B-4. Moves

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0101 1000 xxxx xxxx	movb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0101 1001 xxxx xxxx	movb reg,reg	4	dest	reg	4	src	reg							15	1
0000 0101 xxxx xxxx	movd imm20	20	src	imm	4	dest	rp							12	2
0000 0000 0111 xxxx	movd imm32	4	dest	imm	32	src	rp							23	3
0101 0100 xxxx xxxx	movd imm4/imm16, rp	4	dest	rp	4	src	imm							15/16	1/2
0101 0101 xxxx xxxx	movd rp, rp	4	dest	rp	4	src	rp							15	1
0101 1010 xxxx xxxx	movw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0101 1011 xxxx xxxx	movw reg,reg	4	dest	reg	4	src	reg							15	1
0101 1100 xxxx xxxx	movxb	4	dest	reg	4	src	reg							15	1
0101 1110 xxxx xxxx	movxw	4	dest	rp	4	src	reg							15	1
0101 1101 xxxx xxxx	movzb	4	dest	reg	4	src	reg							15	1
0101 1111 xxxx xxxx	movzw	4	dest	rp	4	src	reg							15	1

Table B-5. Integer Arithmetic

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0011 0000 xxxx xxxx	addb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 0001 xxxx xxxx	addb reg,reg	4	dest	reg	4	src	reg							15	1
0011 0100 xxxx xxxx	addcb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 0101 xxxx xxxx	addcb reg,reg	4	dest	reg	4	src	reg							15	1
0011 0110 xxxx xxxx	addcw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 0111 xxxx xxxx	addcw reg,reg	4	dest	reg	4	src	reg							15	1
0000 0100 xxxx xxxx	addd imm20, rp	20	src	imm	4	dest	rp							12	2
0000 0000 0010 xxxx	addd imm32, rp	4	dest	rp	32	src	imm							23	3
0110 0000 xxxx xxxx	addd imm4/16,rp	4	dest	rp	4	src	imm							15/16	1/2
0110 0001 xxxx xxxx	addd rp,rp	4	dest	rp	4	src	rp							15	1
0010 1100 xxxx xxxx	addub imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 1101 xxxx xxxx	addub reg,reg	4	dest	reg	4	src	reg							15	1
0010 1110 xxxx xxxx	adduw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 1111 xxxx xxxx	adduw reg,reg	4	dest	reg	4	src	reg							15	1
0011 0010 xxxx xxxx	addw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 0011 xxxx xxxx	addw reg,reg	4	dest	reg	4	src	reg							15	1
0000 0000 0001 0100	macqw	4	src2	reg	4	src1	reg	4	dest	rp	4	ope	13	1	2
0000 0000 0001 0100	macuw	4	src2	reg	4	src1	reg	4	dest	rp	4	ope	14	1	2

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0000 0000 0001 0100	macsw	4	src2	reg	4	src1	reg	4	dest	rp	4	ope	15	1	2
0110 0100 xxxx xxxx	mulb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0110 0101 xxxx xxxx	mulb reg,reg	4	dest	reg	4	src	reg							15	1
0000 1011 xxxx xxxx	mulsb reg,reg	4	dest	reg	4	src	reg							15	1
0110 0010 xxxx xxxx	mulsw reg,rp	4	dest	rp	4	src	reg							15	1
0110 0011 xxxx xxxx	muluw reg,rp	4	dest	rp	4	src	reg							15	1
0110 0110 xxxx xxxx	mulw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0110 0111 xxxx xxxx	mulw reg,reg	4	dest	reg	4	src	reg							15	1
0011 1000 xxxx xxxx	subb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 1001 xxxx xxxx	subb reg,reg	4	dest	reg	4	src	reg							15	1
0011 1100 xxxx xxxx	subcb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 1101 xxxx xxxx	subcb reg,reg	4	dest	reg	4	src	reg							15	1
0011 1110 xxxx xxxx	subcw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 1111 xxxx xxxx	subcw reg,reg	4	dest	reg	4	src	reg							15	1
0000 0000 0001 0100	subd rp,rp	4	dest	rp	4	src	rp	4	res	0	4	ope	12	1	2
0000 0000 0011 xxxx	subd imm32,rp	4	dest	rp	32	src	imm							23	3
0011 1010 xxxx xxxx	subw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0011 1011 xxxx xxxx	subw reg,reg	4	dest	reg	4	src	reg							15	1

Table B-6. Integer Comparison

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0101 0000 xxxx xxxx	cmpb imm4/16,reg	4	src	reg	4	src	imm							15/16	1/2
0101 0001 xxxx xxxx	cmpb reg,reg	4	src	reg	4	src	reg							15	1
0000 0000 1001 xxxx	cmpd imm32	4	dest	rp	32	src	imm							23	3
0101 0110 xxxx xxxx	cmpd imm4/16, rp	4	src	rp	4	src	imm							15/16	1/2
0101 0111 xxxx xxxx	cmpd rp, rp	4	src	rp	4	src	rp							15	1
0101 0010 xxxx xxxx	cmpw imm4/16,reg	4	src	reg	4	src	imm							15/16	1/2
0101 0011 xxxx xxxx	cmpw reg,reg	4	src	reg	4	src	reg							15	1

Table B-7. Logical and Boolean

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0010 0000 xxxx xxxx	andb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0010 0001 xxxx xxxx	andb reg,reg	4	dest	reg	4	src	reg							15	1
0010 0010 xxxx xxxx	andw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 0011 xxxx xxxx	andw reg,reg	4	dest	reg	4	src	reg							15	1
0000 0000 0100 xxxx	andd imm32,rp	4	dest	rp	32	src	imm							23	3
0000 0000 0001 0100	andd rp,rp	4	dest	rp	4	src	rp				4	ope	11	1	2
0010 0100 xxxx xxxx	orb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 0101 xxxx xxxx	orb reg,reg	4	dest	reg	4	src	reg							15	1
0010 0110 xxxx xxxx	orw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 0111 xxxx xxxx	orw reg,reg	4	dest	reg	4	src	reg							15	1
0000 0000 0101 xxxx	ord imm32,rp	4	dest	rp	32	src	imm							23	3
0000 0000 0001 0100	ord rp,rp	4	dest	rp	4	src	rp				4	ope	9	1	2
0000 1000 xxxx xxxx	Scond (reg)	4	dest	reg	4	cond	imm							15	1
0010 1000 xxxx xxxx	xorb imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 1001 xxxx xxxx	xorb reg,reg	4	dest	reg	4	src	reg							15	1
0010 1010 xxxx xxxx	xorw imm4/16,reg	4	dest	reg	4	src	imm							15/16	1/2
0010 1011 xxxx xxxx	xorw reg,reg	4	dest	reg	4	src	reg							15	1
0000 0000 0110 xxxx	xord imm32,rp	4	dest	rp	32	src	imm							23	3
0000 0000 0001 0100	xord rp,rp	4	dest	rp	4	src	rp				4	ope	10	1	2

Table B-8. Shifts

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0100 0000 0xxx xxxx	ashub cnt (left +), reg	4	dest	reg	3	count	imm							9	1
0100 0000 1xxx xxxx	ashub cnt (right -), reg	4	dest	reg	3	count	imm							9	1
0100 0001 xxxx xxxx	ashub reg,reg	4	dest	reg	4	count	reg							15	1
0100 110x xxxx xxxx	ashud cnt (left +), rp	4	dest	rp	5	count	imm							20	1
0100 111x xxxx xxxx	ashud cnt (right -), rp	4	dest	rp	5	count	imm							20	1
0100 1000 xxxx xxxx	ashud reg,rp	4	dest	rp	4	count	reg							15	1
0100 0010 xxxx xxxx	ashuw cnt (left +), reg	4	dest	reg	4	count	imm							15	1
0100 0011 xxxx xxxx	ashuw cnt (right -), reg	4	dest	reg	4	count	imm							15	1
0100 0101 xxxx xxxx	ashuw reg,reg	4	dest	reg	4	count	reg							15	1
0000 1001 1xxx xxxx	lshb cnt (right -), reg	4	dest	reg	3	count	imm							9	1
0100 0100 xxxx xxxx	lshb reg,reg	4	dest	reg	4	count	reg							15	1
0100 101x xxxx xxxx	lshd cnt (right -), rp	4	dest	rp	5	count	imm							20	1
0100 0111 xxxx xxxx	lshd reg,rp	4	dest	rp	4	count	reg							15	1

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0100 1001 xxxx xxxx	lshw cnt (right -), reg	4	dest	reg	4	count	imm							15	1
0100 0110 xxxx xxxx	lshw reg,reg	4	dest	reg	4	count	reg							15	1

Table B-9. Bit Operations

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0110 1010 10xx xxxx	cbitb (prp) disp14	4	dest	(prp)	3	pos	imm	14	dest	disp				17	2
0000 0000 0001 0000	cbitb (reg) disp20	4	dest	(reg)	3	pos	imm	20	dest	disp	4	ope	4	2	3
0110 1010 0xxx xxxx	cbitb (rp) disp0	4	dest	(rp)	3	pos	imm							9	1
0110 1011 0xxx xxxx	cbitb (rp) disp16	4	dest	(rp)	3	pos	imm	16	dest	disp				10	2
0000 0000 0001 0000	cbitb (rp) disp20	4	dest	(rp)	3	pos	imm	20	dest	disp	4	ope	5	2	3
0000 0000 0001 0000	cbitb (rrp) disp20	4	dest	(rrp)	3	pos	imm	20	dest	disp	4	ope	6	2	3
0110 1011 1xxx xxxx	cbitb abs20	20	dest	abs	3	pos	imm							7	2
0110 1000 xxxx xxxx	cbitb abs20 rel	20	dest	abs	3	pos	imm	1	dest	rs				8	2
0000 0000 0001 0000	cbitb abs24	24	dest	abs	3	pos	imm				4	ope	7	3	3
0110 1010 11xx xxxx	cbitw (prp) disp14	4	dest	(prp)	4	pos	imm	14	dest	disp				17	2
0000 0000 0001 0001	cbitw (reg) disp20	4	dest	(reg)	4	pos	imm	20	dest	disp	4	ope	4	2	3
0110 1110 xxxx xxxx	cbitw (rp) disp0	4	dest	(rp)	4	pos	imm							15	1
0110 1001 xxxx xxxx	cbitw (rp) disp16	4	dest	(rp)	4	pos	imm	16	dest	disp				16	2
0000 0000 0001 0001	cbitw (rp) disp20	4	dest	(rp)	4	pos	imm	20	dest	disp	4	ope	5	2	3
0000 0000 0001 0001	cbitw (rrp) disp20	4	dest	(rrp)	4	pos	imm	20	dest	disp	4	ope	6	2	3
0110 1111 xxxx xxxx	cbitw abs20	20	dest	abs	4	pos	imm							12	2
0110 110x xxxx xxxx	cbitw abs20 rel	20	dest	abs	4	pos	imm	1	dest	rs				13	2
0000 0000 0001 0001	cbitw abs24	24	dest	abs	4	pos	imm				4	ope	7	3	3
0111 0010 10xx xxxx	sbitb (prp) disp14	4	dest	(prp)	3	pos	imm	14	dest	disp				17	2
0000 0000 0001 0000	sbitb (reg) disp20	4	dest	(reg)	3	pos	imm	20	dest	disp	4	ope	8	2	3
0111 0010 0xxx xxxx	sbitb (rp) disp0	4	dest	(rp)	3	pos	imm							9	1
0111 0011 0xxx xxxx	sbitb (rp) disp16	4	dest	(rp)	3	pos	imm	16	dest	disp				10	2
0000 0000 0001 0000	sbitb (rp) disp20	4	dest	(rp)	3	pos	imm	20	dest	disp	4	ope	9	2	3
0000 0000 0001 0000	sbitb (rrp) disp20	4	dest	(rrp)	3	pos	imm	20	dest	disp	4	ope	10	2	3
0111 0011 1xxx xxxx	sbitb abs20	20	dest	abs	3	pos	imm							7	2
0111 0000 xxxx xxxx	sbitb abs20 rel	20	dest	abs	3	pos	imm	1	dest	rs				8	2
0000 0000 0001 0000	sbitb abs24	24	dest	abs	3	pos	imm				4	ope	11	3	3
0111 0010 11xx xxxx	sbitw (prp) disp14	4	dest	(prp)	4	pos	imm	14	dest	disp				17	2
0000 0000 0001 0001	sbitw (reg) disp20	4	dest	(reg)	4	pos	imm	20	dest	disp	4	ope	8	2	3

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0111 0110 xxxx xxxx	sbitw (rp) disp0	4	dest	(rp)	4	pos	imm							15	1
0111 0001 xxxx xxxx	sbitw (rp) disp16	4	dest	(rp)	4	pos	imm	16	dest	disp				16	2
0000 0000 0001 0001	sbitw (rp) disp20	4	dest	(rp)	4	pos	imm	20	dest	disp	4	ope	9	2	3
0000 0000 0001 0001	sbitw (rrp) disp20	4	dest	(rrp)	4	pos	imm	20	dest	disp	4	ope	10	2	3
0111 0111 xxxx xxxx	sbitw abs20	20	dest	abs	4	pos	imm							12	2
0111 010x xxxx xxxx	sbitw abs20 rel	20	dest	abs	4	pos	imm	1	dest	rs				13	2
0000 0000 0001 0001	sbitw abs24	24	dest	abs	4	pos	imm				4	ope	11	3	3
0000 0110 xxxx xxxx	tbit cnt	4	src	reg	4	pos	imm							15	1
0000 0111 xxxx xxxx	tbit reg, reg	4	src	reg	4	pos	reg							15	1
0111 1010 10xx xxxx	tbitb (prp) disp14	4	dest	(prp)	3	pos	imm	14	dest	disp				17	2
0000 0000 0001 0000	tbitb (reg) disp20	4	dest	(reg)	3	pos	imm	20	dest	disp	4	ope	12	2	3
0111 1010 0xxx xxxx	tbitb (rp) disp0	4	dest	(rp)	3	pos	imm							9	1
0111 1011 0xxx xxxx	tbitb (rp) disp16	4	dest	(rp)	3	pos	imm	16	dest	disp				10	2
0000 0000 0001 0000	tbitb (rp) disp20	4	dest	(rp)	3	pos	imm	20	dest	disp	4	ope	13	2	3
0000 0000 0001 0000	tbitb (rrp) disp20	4	dest	(rrp)	3	pos	imm	20	dest	disp	4	ope	14	2	3
0111 1011 1xxx xxxx	tbitb abs20	20	dest	abs	3	pos	imm							7	2
0111 1000 xxxx xxxx	tbitb abs20 rel	20	dest	abs	3	pos	imm	1	dest	rs				8	2
0000 0000 0001 0000	tbitb abs24	24	dest	abs	3	pos	imm				4	ope	15	3	3
0111 1010 11xx xxxx	tbitw (prp) disp14	4	dest	(prp)	4	pos	imm	14	dest	disp				17	2
0000 0000 0001 0001	tbitw (reg) disp20	4	dest	(reg)	4	pos	imm	20	dest	disp	4	ope	12	2	3
0111 1110 xxxx xxxx	tbitw (rp) disp0	4	dest	(rp)	4	pos	imm							15	1
0111 1001 xxxx xxxx	tbitw (rp) disp16	4	dest	(rp)	4	pos	imm	16	dest	disp				16	2
0000 0000 0001 0001	tbitw (rp) disp20	4	dest	(rp)	4	pos	imm	20	dest	disp	4	ope	13	2	3
0000 0000 0001 0001	tbitw (rrp) disp20	4	dest	(rrp)	4	pos	imm	20	dest	disp	4	ope	14	2	3
0111 1111 xxxx xxxx	tbitw abs20	20	dest	abs	4	pos	imm							12	2
0111 110x xxxx xxxx	tbitw abs20 rel	20	dest	abs	4	pos	imm	1	dest	rs				13	2
0000 0000 0001 0001	tbitw abs24	24	dest	abs	4	pos	imm				4	ope	15	3	3

Table B-10. Processor Register Manipulation

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0000 0000 0001 0100	lpr	4	src	reg	4	dest	pr	4	res	0	4	ope	0	1	2
0000 0000 0001 0100	lprd	4	src	rp	4	dest	prd	4	res	0	4	ope	1	1	2
0000 0000 0001 0100	spr	4	dest	reg	4	src	pr	4	res	0	4	ope	2	1	2
0000 0000 0001 0100	sprd	4	dest	rp	4	src	prd	4	res	0	4	ope	3	1	2

Table B-11. Jumps and Linkage

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt#	ln
1100 0000 xxxx xxxx	bal (ra) disp24	23	dest	disp*2										5	2
0000 0000 0001 0000	bal (rp) disp24	24	dest	disp*2	4	link	rp				4	ope	2	3a	3
0000 1100 xxxx xxxx	beq0b disp4 (2-32)	4	src	reg	4	dest	disp*2+							15	1
0000 1110 xxxx xxxx	beq0w disp4 (2-32)	4	src	reg	4	dest	disp*2+							15	1
0000 1101 xxxx xxxx	bne0b disp4 (2-32)	4	src	reg	4	dest	disp*2+							15	1
0000 1111 xxxx xxxx	bne0w disp4 (2-32)	4	src	reg	4	dest	disp*2+							15	1
0001 xxxx xxxx xxxx	br[a] cond disp8	8	dest	disp*2	4	cond	imm							21	1
0001 1000 xxxx 0000	br[a] cond disp16				4	cond	imm	16	dest	disp*2				22	2
0000 0000 0001 0000	br[a] cond disp24	24	dest	disp*2	4	cond	imm				4	ope	0	3a	3
0000 0000 1100 xxxx	excp	4	vect	imm										11	1
0000 0000 1101 xxxx	jal (ra,rp)	4	dest	rp*2										11	1
0000 0000 0001 0100	jal (rp,rp)	4	link	rp	4	dest	rp*2	*4*	*res*	*0*	4	ope	8	1	2
0000 1010 xxxx xxxx	Jcond[b] (rp)	4	dest	rp*2	4	cond	imm							15	1
0000 0010 xxxx xxxx	pop	4	dest	reg	3	count	imm	1	RA	imm				14	1
0000 0011 xxxx xxxx	pop ret	4	dest	reg	3	count	imm	1	RA	imm				14	1
0000 0001 xxxx xxxx	push	4	src	reg	3	count	imm	1	RA	imm				14	1
0000 0000 0000 0011	retx													4	1

a. This includes BR (condition := always)
b. This includes JUMP (condition := always) and JUSR (condition := UC)

Table B-12. Load and Store

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt#	ln
1011 1110 xxxx xxxx	loadb (prp) disp0	4	src	(prp)	4	dest	reg	4	ope	0E				18	1
1000 0110 01xx xxxx	loadb (prp) disp14	4	src	(prp)	4	dest	reg	14	src	disp				17	2
0000 0000 0001 0010	loadb (reg) disp20	4	src	(reg)	4	dest	reg	20	src	disp	4	ope	4	2	3
0000 0000 0001 1000	loadb (reg) -disp20	4	src	(reg)	4	dest	reg	20	src	disp	4	ope	4	2	3
1011 1111 xxxx xxxx	loadb (rp) disp16	4	src	(rp)	4	dest	reg	4	ope	0F	16	src	disp	19	2
0000 0000 0001 0010	loadb (rp) disp20	4	src	(rp)	4	dest	reg	20	src	disp	4	ope	5	2	3
0000 0000 0001 1000	loadb (rp) -disp20	4	src	(rp)	4	dest	reg	20	src	disp	4	ope	5	2	3
1011 xxxx xxxx xxxx	loadb (rp) disp4	4	src	(rp)	4	dest	reg	4	src	disp				18	1
0000 0000 0001 0010	loadb (rrp) disp20	4	src	(rrp)	4	dest	reg	20	src	disp	4	ope	6	2	3
1000 1000 xxxx xxxx	loadb abs20	20	src	abs	4	dest	reg							12	2
1000 101x xxxx xxxx	loadb abs20 rel	20	src	abs	4	dest	reg	1	src	rs				13	2
0000 0000 0001 0010	loadb abs24	24	src	abs	4	dest	reg				4	ope	7	3	3
1010 1110 xxxx xxxx	loadd (prp) disp0	4	src	(prp)	4	dest	rp	4	ope	0E				18	1

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt#	ln
1000 0110 10xx xxxx	loadd (prp) disp14	4	src	(prp)	4	dest	rp	14	src	disp				17	2
0000 0000 0001 0010	loadd (reg) disp20	4	src	(reg)	4	dest	rp	20	src	disp	4	ope	8	2	3
0000 0000 0001 1000	loadd (reg) -disp20	4	src	(reg)	4	dest	rp	20	src	disp	4	ope	8	2	3
1010 1111 xxxx xxxx	loadd (rp) disp16	4	src	(rp)	4	dest	rp	4	ope	0F	16	src	disp	19	2
0000 0000 0001 0010	loadd (rp) disp20	4	src	(rp)	4	dest	rp	20	src	disp	4	ope	9	2	3
0000 0000 0001 1000	loadd (rp) -disp20	4	src	(rp)	4	dest	rp	20	src	disp	4	ope	9	2	3
1010 xxxx xxxx xxxx	loadd (rp) disp4	4	src	(rp)	4	dest	rp	4	src	disp*2				18	1
0000 0000 0001 0010	loadd (rrp) disp20	4	src	(rrp)	4	dest	rp	20	src	disp	4	ope	10	2	3
1000 0111 xxxx xxxx	loadd abs20	20	src	abs	4	dest	rp							12	2
1000 110x xxxx xxxx	loadd abs20 rel	20	src	abs	4	dest	rp	1	src	rs				13	2
0000 0000 0001 0010	loadd abs24	24	src	abs	4	dest	rp				4	ope	11	3	3
0000 0000 1010 0xxx	loadm (reg)	3	count	imm										6	1
0000 0000 1010 1xxx	loadmp (rp)	3	count	imm										6	1
1001 1110 xxxx xxxx	loadw (prp) disp0	4	src	(prp)	4	dest	reg	4	ope	0E				18	1
1000 0110 11xx xxxx	loadw (prp) disp14	4	src	(prp)	4	dest	reg	14	src	disp				17	2
0000 0000 0001 0010	loadw (reg) disp20	4	src	(reg)	4	dest	reg	20	src	disp	4	ope	12	2	3
0000 0000 0001 1000	loadw (reg) -disp20	4	src	(reg)	4	dest	reg	20	src	disp	4	ope	12	2	3
1001 1111 xxxx xxxx	loadw (rp) disp16	4	src	(rp)	4	dest	reg	4	ope	0F	16	src	disp	19	2
0000 0000 0001 0010	loadw (rp) disp20	4	src	(rp)	4	dest	reg	20	src	disp	4	ope	13	2	3
0000 0000 0001 1000	loadw (rp) -disp20	4	src	(rp)	4	dest	reg	20	src	disp	4	ope	13	2	3
1001 xxxx xxxx xxxx	loadw (rp) disp4	4	src	(rp)	4	dest	reg	4	src	disp*2				18	1
0000 0000 0001 0010	loadw (rrp) disp20	4	src	(rrp)	4	dest	reg	20	src	disp	4	ope	14	2	3
1000 1001 xxxx xxxx	loadw abs20	20	src	abs	4	dest	reg							12	2
1000 111x xxxx xxxx	loadw abs20 rel	20	src	abs	4	dest	reg	1	src	rs				13	2
0000 0000 0001 0010	loadw abs24	24	src	abs	4	dest	reg				4	ope	15	3	3
1111 1110 xxxx xxxx	storb (prp) disp0	4	dest	(prp)	4	src	reg	4	ope	0E				18	1
1100 0110 01xx xxxx	storb (prp) disp14	4	dest	(prp)	4	src	reg	14	dest	disp				17	2
0000 0000 0001 0011	storb (reg) disp20	4	dest	(reg)	4	src	reg	20	dest	disp	4	ope	4	2	3
0000 0000 0001 1001	storb (reg) -disp20	4	dest	(reg)	4	src	reg	20	dest	disp	4	ope	4	2	3
1111 1111 xxxx xxxx	storb (rp) disp16	4	dest	(rp)	4	src	reg	4	ope	0F	16	dest	disp	19	2
0000 0000 0001 0011	storb (rp) disp20	4	dest	(rp)	4	src	reg	20	dest	disp	4	ope	5	2	2
0000 0000 0001 1001	storb (rp) -disp20	4	dest	(rp)	4	src	reg	20	dest	disp	4	ope	5	2	2
1111 xxxx xxxx xxxx	storb (rp) disp4[a]	4	dest	(rp)	4	src	reg	4	dest	disp				18	1
0000 0000 0001 0011	storb (rrp) disp20	4	dest	(rrp)	4	src	reg	20	dest	disp	4	ope	6	2	2
1100 1000 xxxx xxxx	storb abs20	20	dest	abs	4	src	reg							12	2
1100 101x xxxx xxxx	storb abs20 rel	20	dest	abs	4	src	reg	1	dest	rs				13	2

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt#	ln
0000 0000 0001 0011	storb abs24	24	dest	abs	4	src	reg				4	ope	7	3	3
1000 0110 00xx xxxx	storb imm (prp) disp14	4	dest	(prp)	4	src	imm	14	dest	disp				17	2
0000 0000 0001 0010	storb imm (reg) disp20	4	dest	(reg)	4	src	imm	20	dest	disp	4	ope	0	2	3
1000 0010 xxxx xxxx	storb imm (rp) disp0	4	dest	(rp)	4	src	imm							15	1
1000 0011 xxxx xxxx	storb imm (rp) disp16	4	dest	(rp)	4	src	imm	16	dest	disp				16	2
0000 0000 0001 0010	storb imm (rp) disp20	4	dest	(rp)	4	src	imm	20	dest	disp	4	ope	1	2	3
0000 0000 0001 0010	storb imm (rrp) disp20	4	dest	(rrp)	4	src	imm	20	dest	disp	4	ope	2	2	3
1000 0001 xxxx xxxx	storb imm abs20	20	dest	abs	4	src	imm							12	2
1000 010x xxxx xxxx	storb imm abs20 rel	20	dest	abs	4	src	imm	1	dest	rs				13	2
0000 0000 0001 0010	storb imm abs24	24	dest	abs	4	src	imm				4	ope	3	3	3
1110 1110 xxxx xxxx	stord (prp) disp0	4	dest	(prp)	4	src	rp	4	ope	0E				18	1
1100 0110 10xx xxxx	stord (prp) disp14	4	dest	(prp)	4	src	rp	14	dest	disp				17	2
0000 0000 0001 0011	stord (reg) disp20	4	dest	(reg)	4	src	rp	20	dest	disp	4	ope	8	2	3
0000 0000 0001 1001	stord (reg) -disp20	4	dest	(reg)	4	src	rp	20	dest	disp	4	ope	8	2	3
1110 1111 xxxx xxxx	stord (rp) disp16	4	dest	(rp)	4	src	rp	4	ope	0F	16	dest	disp	19	2
0000 0000 0001 0011	stord (rp) disp20	4	dest	(rp)	4	src	rp	20	dest	disp	4	ope	9	2	3
0000 0000 0001 1001	stord (rp) -disp20	4	dest	(rp)	4	src	rp	20	dest	disp	4	ope	9	2	3
1110 xxxx xxxx xxxx	stord (rp) disp4	4	dest	(rp)	4	src	rp	4	dest	disp*2				18	1
0000 0000 0001 0011	stord (rrp) disp20	4	dest	(rrp)	4	src	rp	20	dest	disp	4	ope	10	2	3
1100 0111 xxxx xxxx	stord abs20	20	dest	abs	4	src	rp							12	2
1100 110x xxxx xxxx	stord abs20 rel	20	dest	abs	4	src	rp	1	dest	rs				13	2
0000 0000 0001 0011	stord abs24	24	dest	abs	4	src	rp				4	ope	11	3	3
0000 0000 1011 0xxx	storm (reg)	3	count	imm										6	1
0000 0000 1011 1xxx	stormp (rp)	3	count	imm										6	1
1101 1110 xxxx xxxx	storw (prp) disp0	4	dest	(prp)	4	src	reg	4	ope	0E				18	1
1100 0110 11xx xxxx	storw (prp) disp14	4	dest	(prp)	4	src	reg	14	dest	disp				17	2
0000 0000 0001 0011	storw (reg) disp20	4	dest	(reg)	4	src	reg	20	dest	disp	4	ope	12	2	3
0000 0000 0001 1001	storw (reg) -disp20	4	dest	(reg)	4	src	reg	20	dest	disp	4	ope	12	2	3
1101 1111 xxxx xxxx	storw (rp) disp16	4	dest	(rp)	4	src	reg	4	ope	0F	16	dest	disp	19	2
0000 0000 0001 0011	storw (rp) disp20	4	dest	(rp)	4	src	reg	20	dest	disp	4	ope	13	2	3
0000 0000 0001 1001	storw (rp) -disp20	4	dest	(rp)	4	src	reg	20	dest	disp	4	ope	13	2	3
1101 xxxx xxxx xxxx	storw (rp) disp4	4	dest	(rp)	4	src	reg	4	dest	disp*2				18	1
0000 0000 0001 0011	storw (rrp) disp20	4	dest	(rrp)	4	src	reg	20	dest	disp	4	ope	14	2	3
1100 1001 xxxx xxxx	storw abs20	20	dest	abs	4	src	reg							12	2
1100 111x xxxx xxxx	storw abs20 rel	20	dest	abs	4	src	reg	1	dest	rs				13	2
0000 0000 0001 0011	storw abs24	24	dest	abs	4	src	reg				4	ope	15	3	3

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt#	ln
1100 0110 00xx xxxx	storw imm (prp) disp14	4	dest	(prp)	4	src	imm	14	dest	disp				17	2
0000 0000 0001 0011	storw imm (reg) disp20	4	dest	(reg)	4	src	imm	20	dest	disp	4	ope	0	2	3
1100 0010 xxxx xxxx	storw imm (rp) disp0	4	dest	(rp)	4	src	imm							15	1
1100 0011 xxxx xxxx	storw imm (rp) disp16	4	dest	(rp)	4	src	imm	16	dest	disp				16	2
0000 0000 0001 0011	storw imm (rp) disp20	4	dest	(rp)	4	src	imm	20	dest	disp	4	ope	1	2	3
0000 0000 0001 0011	storw imm (rrp) disp20	4	dest	(rrp)	4	src	imm	20	dest	disp	4	ope	2	2	3
1100 0001 xxxx xxxx	storw imm abs20	20	dest	abs	4	src	imm							12	2
1100 010x xxxx xxxx	storw imm abs20 rel	20	dest	abs	4	src	imm	1	dest	rs				13	2
0000 0000 0001 0011	storw imm abs24	24	dest	abs	4	src	imm				4	ope	3	3	3

Table B-13. Miscellaneous

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0000 0000 0000 1010	cinv [i]													4	1
0000 0000 0000 1011	cinv [i,u]													4	1
0000 0000 0000 1100	cinv [d]													4	1
0000 0000 0000 1101	cinv [d,u]													4	1
0000 0000 0000 1110	cinv [d,i]													4	1
0000 0000 0000 1111	cinv [d,i,u]													4	1
0000 0000 0000 0100	di													4	1
0000 0000 0000 0101	ei													4	1
0000 0000 0000 0111	eiwait													4	1
0000 0000 0000 0110	wait													4	1

Table B-14. Reserved (res)

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0000 0000 0000 000x	res - undefined trap														1
0000 0000 0000 0010	res - undefined trap														1
0000 0000 0000 100x	res - undefined trap														1
0000 0000 0001 0000	res - no operation										4	ope	1	3	3
0000 0000 0001 0000	res - no operation										4	ope	3	3	3
0000 0000 0001 0001	res - no operation										4	ope	0	3	3
0000 0000 0001 0001	res - no operation										4	ope	1	3	3
0000 0000 0001 0001	res - no operation										4	ope	2	3	3
0000 0000 0001 0001	res - no operation										4	ope	3	3	3
0000 0000 0001 0100	res - no operation										4	ope	4	1	2
0000 0000 0001 0100	res - no operation										4	ope	5	1	2
0000 0000 0001 0100	res - no operation										4	ope	6	1	2
0000 0000 0001 0100	res - no operation										4	ope	7	1	2
0000 0000 0001 0101	res - undefined trap													1	2
0000 0000 0001 0110	res - undefined trap for movmcr (rp)	*4*	*dest*	*rp*	*3*	*count*	*imm*	*4*	*src*	*ci*	*4*	*src*	*ca*	*1*	*2*
0000 0000 0001 0111	res - undefined trap for movmrc (rp)	*4*	*src*	*rp*	*3*	*count*	*imm*	*4*	*dest*	*ci*	*4*	*dest*	*ca*	*1*	*2*
0000 0000 0001 1000	res - undefined trap											ope	0		3
0000 0000 0001 1000	res - undefined trap											ope	1		3
0000 0000 0001 1000	res - undefined trap											ope	2		3
0000 0000 0001 1000	res - undefined trap											ope	3		3
0000 0000 0001 1000	res - undefined trap											ope	6		3
0000 0000 0001 1000	res - undefined trap											ope	7		3
0000 0000 0001 1000	res - undefined trap											ope	10		3
0000 0000 0001 1000	res - undefined trap											ope	11		3
0000 0000 0001 1000	res - undefined trap											ope	14		3
0000 0000 0001 1000	res - undefined trap											ope	15		3
0000 0000 0001 1001	res - undefined trap											ope	0		3
0000 0000 0001 1001	res - undefined trap											ope	1		3
0000 0000 0001 1001	res - undefined trap											ope	2		3
0000 0000 0001 1001	res - undefined trap											ope	3		3
0000 0000 0001 1001	res - undefined trap											ope	6		3
0000 0000 0001 1001	res - undefined trap											ope	7		3
0000 0000 0001 1001	res - undefined trap											ope	10		3

Opcode (15:0)	Instruction Name	p1	pt1	pm1	p2	pt2	pm2	p3	pt3	pm3	p4	pt4	pm4	fmt #	ln
0000 0000 0001 1001	res - undefined trap											ope	11		3
0000 0000 0001 1001	res - undefined trap											ope	14		3
0000 0000 0001 1001	res - undefined trap											ope	15		3
0000 0000 0001 101x	res - undefined trap														3
0000 0000 0001 11xx	res - undefined trap														3
0000 0000 1000 xxxx	res - undefined trap *for coprocessor inst*	*4*	*ci*	*imm*	*32*	*cinst*	*imm*							23	3
0000 0000 111x xxxx	res - undefined trap														1
0000 1001 0xxx xxxx	res - undefined trap														2
1000 0000 xxxx xxxx	res - undefined trap													15	1
1100 0000 xxxx xxxx	res - undefined trap													15	1
1111 1111 1111 1111	res - undefined trap														1

This page is intentionally blank.

Appendix C
STANDARD CALLING CONVENTIONS

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 CompactRISC development toolsets. 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.

If you are using a toolset other than National's CompactRISC Toolset, it might support another calling convention, in addition to the one below. Please refer to your toolset's manual.

C.1 CALLING A ROUTINE

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.

C.2 SUBROUTINES

C.2.1 Calling a Subroutine

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

1. It saves the address of the following instruction (the value of the Program Counter Register) in a specified general-purpose register pair.

   The calling convention requires the use of (ERA, RA) register pair (when CFG.SR=1), or the RA Register (CFG.SR =0). This register pair is used to store the return address.

2. It transfers control to a specified location in the program (the subroutine address).

Example 1 For CFG.SR=1:

```
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
```

C.2.2 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, or the RA Register, as follows:

For CFG.SR=1:

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

C.3 CALLING CONVENTION ELEMENTS

This section describes the conventions for passing parameters to a called subroutine and for returning a value. It also discusses the program stack and how to call different types of general-purpose registers.

C.3.1 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.3.2 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 CFG.SR=1:

```
movw $5, r0      # pass return value
jump (era,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.3.3 Program Stack

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

- Allocating memory for local variables which are not in registers
- Passing arguments in special cases (see Section C.3.1 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.

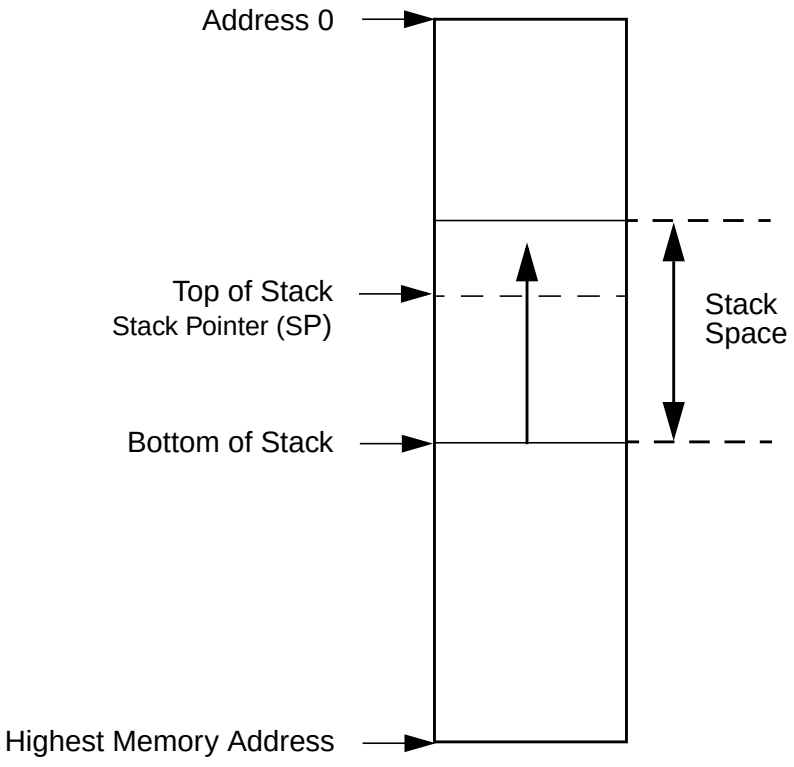


Figure C-1. The Program Stack

A subroutine can allocate space on the stack by decrementing the value of the SP 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 lifetime.

C.3.4 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, the scratch register must be saved on the stack before calling the subroutine and restored 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 used by any subroutine that it has called.

When CFG.SR = 1, the calling convention defines R0 through R6 as scratch registers. All 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.

This page is intentionally blank.

Appendix D
COMPARING CR16C WITH CR16A/B

The purpose of this appendix is to help CR16A and CR16B developers migrate to the CR16C. It focuses on both the implications for applications using the large programming model, and on the differences between the programming model of the CR16B and the CR16C, particularly on the instruction set differences.

Knowledge of these differences can enhance performance and enable code optimization of existing applications.

D.1 MAJOR ENHANCEMENTS FROM CR16A/B

The CR16C architecture is an enhancement of the CR16B architecture. The major enhancement is an increase of the available address space. In the initial implementation of the CR16C, a 24-bit address is supported allowing a 16M address space. The increased address size impacts on the instruction encoding and register usage.

The instruction encoding has been changed to accommodate additional commands that allow more efficient handling of larger address pointers. As a result, binary compatibility with CR16A/B code is no longer maintained. However, assembly level compatibility is supported.

In addition, to offset the additional register usage required to hold larger address pointers, registers R12-R15 are doubled in size.

To simplify backward assembly-level compatibility with the CR16A and CR16B, the CR16C provides a configuration bit, CFG.SR, that permits exclusive use of only the small registers. The small programming model of the CR16B, which is backward compatible with the CR16A, is not supported directly.

D.2 REGISTER SET

See Figures D-1 and D-2 for CR16B and CR16C register mapping.

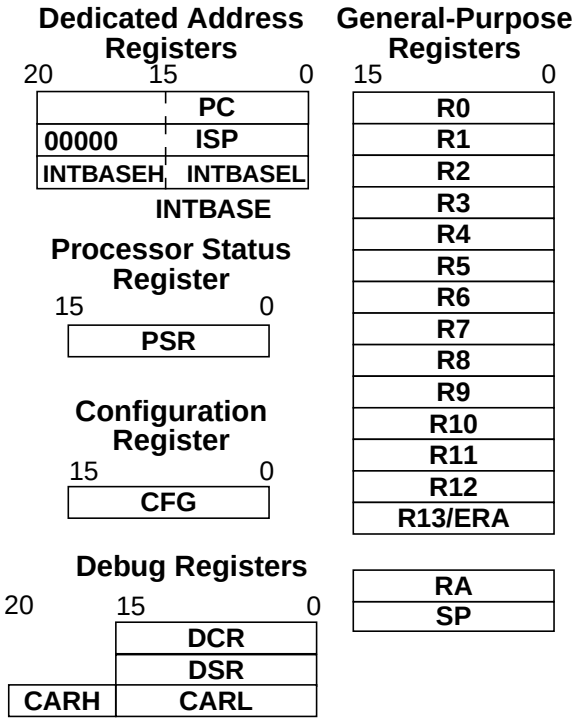


Figure D-1. CR16B Register Set

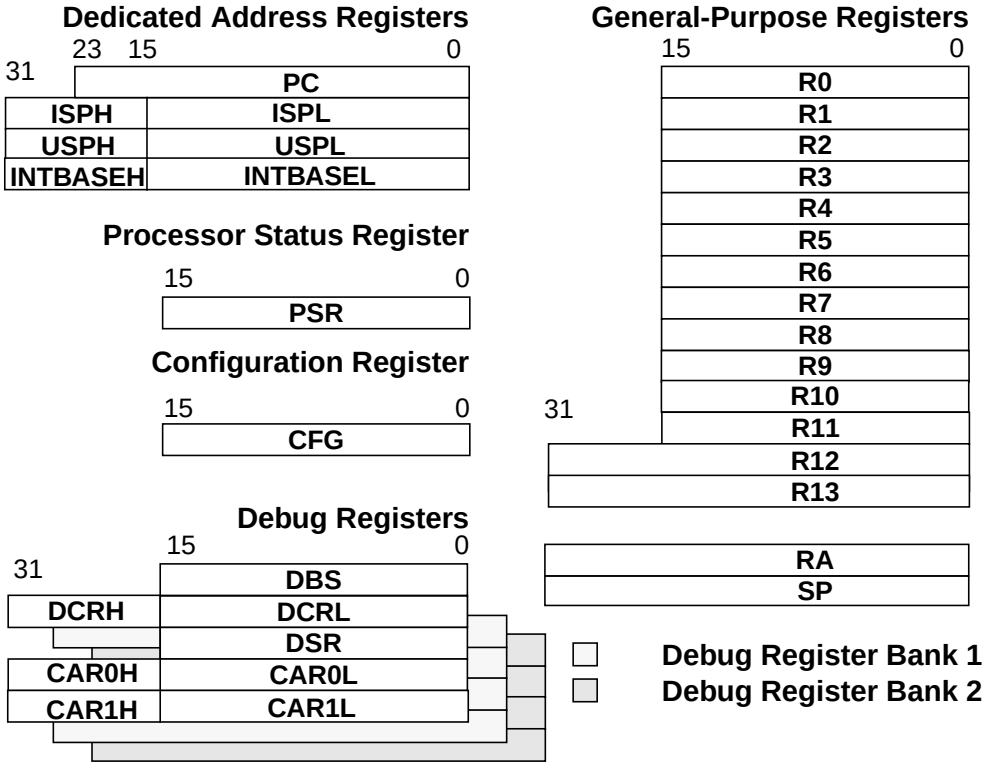


Figure D-2. CR16C Programming Model

D.2.1 General-Purpose Registers

The general-purpose registers are 16 bits wide in the CR16A and the CR16B. In the CR16C, registers R12, R13, RA and SP are doubled in size to 32 bits. In the CR16C, R12 and R13 can be used as base and index address registers as a single register. In addition, all other registers can be paired.

D.2.2 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 the CR16B. In the CR16C, these registers are increased to 32 bits. See Section 2.3.2 on page 2-3 for a detailed description of these registers.

CR16B large programming model

In the CR16B, the three dedicated address registers, PC, ISP and INTABSE, are 21 bits wide. However, the five most significant bits of the **ISP** registers are always cleared. In the CR16C, these registers are all 24 bits.

Configuration bits

A short register bit allows use of the registers as in the CR16B; the extended version of R12, R13 and R14 is effectively disabled.

In the Debug Control Register, there is also an option to use a separate Compare Address Register, either independently or as the second part of a range compare.

In addition, support for a cache is included with instruction and data cache enable bits.

PSR

A user mode status bit has been added to indicate whether the processor is currently in user mode, or is using the user stack.

D.3 INSTRUCTION SET

New instructions

After extensive code analysis, new instructions have been added to support additional operations that work on a larger address space. The following new commands accommodate the larger address pointers:

- LOAD/STORE double-word
- Compare double
- Add double
- Subtract double
- Arithmetic/Logical shift double
- MOVXW/MOVZW
- AND/OR/XOR double

In response to user feedback, and to facilitate simple DSP algorithms, a basic set of Multiply and Accumulate instructions has been included in the instruction set:

- MACSW
- MACUW
- MACQW

Relative addressing

Addressing relative to a register pair has been expanded to allow any register pair as the base address, across the entire address range. Direct access using 20 and 24-bit labels is also provided across all data operations.

Index addressing mode

Improved support for relocatable code and index addressing mode has been added. This allows the location of data to be determined at run-time. Code is written only with relative jumps and data may be located in two regions; for example, one for the RAM and one for flash.

Expanded instructions

The following instructions have an expanded meaning:

PUSH/POP/POPRET now operate on up to eight registers with a separate bit for determining if RA should be pushed/popped to/from the stack. This avoids two consecutive stack operations, which often occur.

Load/Store Multiple now allows a count of up to eight registers, and permits a register pair to contain the source and destination. The former **Load/Store Multiple** instruction, which uses a single register pointer, is still supported but now with a count of eight.

All restrictions have been removed from **bit** and **Store Immediate** operations regarding which registers can be used to store the address displacements. Additional addressing modes are provided.

Retired instructions

Since the CR16B small programming model is no longer supported directly, the following instructions are no longer necessary:

- **bal** (reg) disp17
- **jal** (reg,reg)
- **Jcond** (reg)
- **jump** (reg)
- **POPRET** small model

The **Bcond1i** instruction is no longer implemented by the CR core. The **Bcond1i** assembly instruction is converted by the assembler to a **cmp** and bcond instruction.

Peripheral and interrupt mapping

Peripherals can be mapped into the last 64K of the address space. This allows RAM to start at address 0 and increase as much as necessary, without requiring holes in the memory decode logic. For coding effi-

ciency, access to this address space can be accomplished with the 20-bit absolute addressing mode as double-word commands by mapping the last 64K in the first MB of memory to the last 64K of the 16M space.

D.4 EXCEPTION HANDLING

Dispatch table

The dispatch table no longer has any location restrictions. It can exist anywhere in memory. The CFG.ED bit now determines only if it contains single-word or double-word entries. This allows size optimization of the dispatch table, since on-chip RAM is always at a premium.

Interrupt stack

Due to the larger address pointers, an entry on the interrupt stack must now be in 3-word format, including: PSR, address high, address low.

Wraparound

CR16A and CR16B address wraparound is no longer supported. An address out of the range of 0 to 16M triggers an IA trap.

User mode

The CR16C adds better support for multi-tasking operating systems, and now supports a user stack in addition a supervisor stack. This is a requirement for many secure applications. An application that does not use the user stack runs entirely in supervisor mode. All interrupts and traps are handled in supervisor mode.

Two stack pointers are used. The user stack pointer is initialized with LPR commands. The **JUSR** instruction causes entry to user mode by setting the PSR.U bit. This bit is output from the core on the SFUSR signal for use by an external Memory Protection Unit. This bit is also used to determine which current stack pointer to use. The PSR.U bit is cleared on an exception before the dispatch table is read.

Table D-1. Execution Times for Interrupts and Traps

Interrupt or Trap	Clock Cycles in EX Stage	
	CFG.ED=0 (16-bit dispatch table)	CFG.ED=1 (32-bit dispatch table)
INT	12	13
NMI and ISE[a]	11	12
TRC[a]	9	10
DBG[a]	9	10
SVC, DVZ, FLG, BPT[a] and UND	9	10

a. For ISE, TRC, DBG and BPT, if AISE, ATRC, ADBG and ABPT bits in the CFG registers are set (respectively), then the execution time is that of ISE/NMI for Large Model the 32-bit dispatch table (12 cycles). See Section 4.1.5 on page 4-14 for more details.

This page is intentionally blank.

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z