Texas Instruments

TMS370C16 Central Processing Unit, System, and Instruction Set

PRISM Module Library

Reference Guide

1994

cMCU370™ Products

TMS370C16 Central Processing Unit, System, and Instruction Set Reference Guide

PRISM Module Library

March 1994

Texas Instruments

IMPORTANT NOTICE

Texas Instruments Incorporated (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

Overview

Texas Instruments uses PRISM methodology, with its modular fabrication processes, to integrate analog and digital functions on a single chip. The process technologies currently include VLSI CMOS, nonvolatile memories (EPROM/EEPROM), lateral DMOS, high-voltage analog CMOS, and high-density analog CMOS

The 16-bit TMS370C16 CPU is part of the cMCU370™ family of microcontroller devices. This manual provides information about the TMS370C16 CPU architecture, features, operation, and assembly language instruction set; it also includes helpful information about implementing a TMS370C16-based microcontroller design.

Related documentation is listed on page v.

Manual Organization

- **Chapter 1** gives a brief overview of the TMS370C16 microcontroller device.
- **Chapter 2** describes the components and operation of the TMS370C16 CPU architecture, including CPU registers and memory organization.
- **Chapter 3** describes the TMS370C16 system configuration, registers, device interrupts, and reset.
- **Chapter 4** describes the different addressing modes used by the instruction set.
- **Chapter 5** lists and describes the TMS370C16 assembly language instructions, execution sequence, effects, and examples.
- **Appendix A, Glossary,** explains and defines terms and abbreviations used in this manual.

Style, Symbols, and Definitions

This document uses the following conventions.

- Abbreviations:
  - **'C16**: TMS370C16 CPU-based devices
  - **LSB, MSB**: Least significant and most significant *bits*
  - **LSbyte, MSbyte**: Least and most significant *bytes*
  - Register and bit names: **SCR1.7**, for example

    The register name (located to the left of the period) is an alpha abbreviation (e.g., SSR = system status register, and SCR1 = system control register 1). The bit number is to the right of the period (e.g., SCR1.7 is bit 7 of register SCR1 as shown in Figure 3–3 on page 3-7).

- Definitions of *device* and *module* as used in this manual:
  - **Device:** The cMCU370 microcontroller; includes the TMS370C16 CPU along with all selected modules integrated on a single chip.
  - **Module:** An element that provides a specific function (such as a serial interface, memory, analog-to-digital conversion, timing, I/0, etc.). A list of modules is provided in the documentation-title list on page v (in this preface).

- Program listings and program examples are shown in a `special typeface` similar to a typewriter's.

Note: Assembler Statements Are Not Case Sensitive

TMS370C16 assembly language statements are not case sensitive. You can enter them in lowercase, uppercase, or a combination. To emphasize this, assembly language statements are shown throughout this user's guide in both uppercase and lowercase.

Related Documentation From Texas Instruments

	Literature Number
❑ TMS370C8 CPU, System, and Instruction Set Reference Guide	SPNU042
❑ TMS370C16 CPU, System, and Instruction Set Reference Guide	SPNU043
❑ PRISM Module Library Reference Set, Volume 1	SPNU031

Volume 1 includes the following module reference guides.

- cMCU370 Microcontroller Products Introduction
- Clock Modules Reference Guide
- Watchdog and Real-Time Interrupt Module Reference Guide
- EEPROM/EPROM Modules Reference Guide
- TMS370C8 Timer Modules Reference Guide
- Serial Communications Interface Module Reference Guide
- Serial Peripheral Interface Module Reference Guide
- Analog-to-Digital Converter Module Reference Guide

❑ PRISM Module Library Reference Set, Volume 2 SPNU032

Volume 2 includes the following module reference guides.

- TMS370C16 Timer Modules Reference Guide
- Voltage Regulator Modules Reference Guide
- Gage Driver Modules Reference Guide
- Power Driver Modules Reference Guide
- Switch Interface Module Reference Guide
- Variable Reluctance Sensor Module Reference Guide

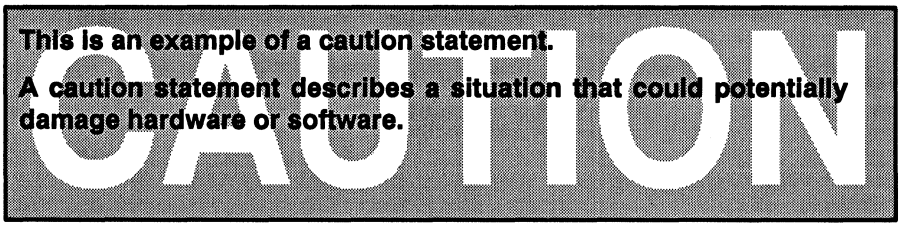
Some books on this list will be available at a later date.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Ask questions about product operation, or report suspected problems	Call the TI microcontroller hotline: **(713) 274–2370** **FAX: (713) 274–4203**
Request more information about Texas Instruments products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 6101 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Bulletin board number	**(713) 274–3700**
Report mistakes in this document or any other TI documentation	Send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443 **or** call the TI microcontroller hotline (phone number at top of this table)

Information About Cautions

The information in a caution is provided for your protection. Please read each caution carefully.



Trademarks

cMCU370 is a registered trademark of Texas Instruments Incorporated.

Contents

Figures

Tables

Notes and Cautions

Chapter 1

Introduction

The TMS370C16 microcontroller core is part of the PRISM Modular Library. With reusable engineering techniques, it can be combined with other building blocks from the modular library to generate a diversified family of highly integrated devices.

This chapter gives a brief overview of the 'C16 CPU — its device-specific operation, its features, and its registers.

This chapter covers the following topics:

Topic **Page**

1.1 TMS370C16 CPU — Device-Specific Operation

The total integration concept of the cMCU microcontroller family makes multiple configurations possible. Because of this flexibility, certain module features are device specific and therefore cannot be presented as an absolute in this document. You should refer to the specific device data sheet to determine the features and functions available on your particular device. Here is a partial list of these indefinable areas:

- ❑ Memory array size and memory map location for RAM, ROM/EPROM, EEPROM, and peripheral file
- ❑ System clock (SYSCLK) operation
- ❑ Digital I/O pin functionality
- ❑ Interrupts (The number of available external and internal interrupts and their associated vectors.)
- ❑ Low-power mode availability and interrupt exit capability.

Note: Definitions of Device and Module Used In This Manual

Device: The core microcontroller. It includes the CPU (TMS370C16), along with all selected modules, integrated on a single chip.

Module: An element that provides a specific function (such as a serial interface, memory, analog-to-digital conversion, timing, I/0, etc.) A list of modules is provided on page v of the preface.

1.2 CPU, System, and Instruction Set Features

The TMS370C16 CPU module consists of the following:

- 16-bit CPU containing the associated registers:
  - Frame pointer
  - Implied register
  - Stack pointer
  - Status register
  - Zero register
  - 16-bit program counter
- 17-bit address space
- Various memory types supported by the 'C16 architecture
  - RAM
  - Peripheral file control registers
  - Data EEPROM
  - Program memory (ROM or EPROM)
- Seven possible reset sources
- Interrupt structure
  - Software-selectable priority levels
  - Nonmaskable Interrupt (NMI) options
  - Variable number of interrupts, depending on the device configurations
  - Individual interrupt vectors
- Two low-power modes
- Set of 126 instructions including byte, word, and long-word formats.

1.3 TMS370C16 Control Registers

The CPU and system functions are controlled by registers in three separate frames as illustrated in the following three tables.

Table 1–1. TMS370C16 System Configuration Control Registers

Address	Register Symbol	Register Name	Described In Section	Described In Page
0010h		*Reserved*		
to		•		
0017h		*Reserved*		
0018h	SCR0	System Control Register 0	3.5.1	3-8
0019h	SCR1	System Control Register 1	3.5.2	3-9
001Ah	SRSR	System Reset Status Register	3.5.3	3-10
001Bh	SSR	System Status Register	3.5.4	3-12
001Ch		*Reserved*		
001Dh		*Reserved*		
001Eh	PSAR1	Parallel Signature Analysis Register 1	3.5.5	3-13
001Fh	PSAR2	Parallel Signature Analysis Register 2	3.5.5	3-13

Table 1–2. TMS370C16 Digital Pin Function Control Registers

Address	Register Symbol	Register Name	Described In Section	Described In Page
0060h	OCR1	Output/Control Register 1	3.6.1	3-16
0061h	OCR2	Output/Control Register 2	3.6.1	3-16
0062h	OCR3	Output/Control Register 3	3.6.1	3-16
0063h	OCR4	Output/Control Register 4	3.6.1	3-16
0064h	ISR1	Input/Status Register 1	3.6.2	3-16
0065h	ISR2	Input/Status Register 2	3.6.2	3-16
0066h	ISR3	Input/Status Register 2	3.6.2	3-16
0067h	ISR4	Input/Status Register 2	3.6.2	3-16
0068h	ADIR	I/O Port A Direction Register	3.6.3	3-17
0069h	ADATA	I/O Port A Data Register	3.6.3	3-17
006Ah	BDIR	I/O Port B Direction Register	3.6.3	3-17
006Bh	BDATA	I/O Port B Data Register	3.6.3	3-17
006Ch	CDIR	I/O Port C Direction Register	3.6.3	3-17
006Dh	CDATA	I/O Port C Data Register	3.6.3	3-17
006Eh	DDIR	I/O Port D Direction Register	3.6.3	3-17
006Fh	DDATA	I/O Port D Data Register	3.6.3	3-17

Table 1–3. TMS370C16 Typical Interrupt Control Registers

Address	Register Symbol	Register Name	Described in Section	Described in Page
0070h	INT1	Type A Interrupt	3.8.1.1	3-29
0071h	INT1 FLG	Type A Interrupt Flag	3.8.1.1	3-29
0072h	INT2	Type B Interrupt	3.8.1.3	3-31
0073h	INT2 FLG	Type B Interrupt Flag	3.8.1.3	3-31
0074h	INT3	Type C Interrupt	3.8.1.5	3-33
0075h	INT3 FLG	Type C Interrupt Flag	3.8.1.5	3-33
0076h		*Reserved*		
0077h		*Reserved*		
0078h		*Reserved*		
0079h		*Reserved*		
007Ah		*Reserved*		
007Bh		*Reserved*		
007Ch	PM2 ENABLE	Power Module Interrupt Enable Register 2	3.8.2.1	3-35
007Dh	PM2 FLAGS	Power Module Interrupt Flag Register 2	3.8.2.2	3-36
007Eh	PM1 ENABLE	Power Module Interrupt Enable Register 1	3.8.2.1	3-35
007Fh	PM1 FLAGS	Power Module Interrupt Flag Register 1	3.8.2.2	3-36

Chapter 2

Architecture

This chapter describes the programmer's model registers and how the 128K-byte memory is organized and addressed. Topics in this chapter include:

Topic **Page**

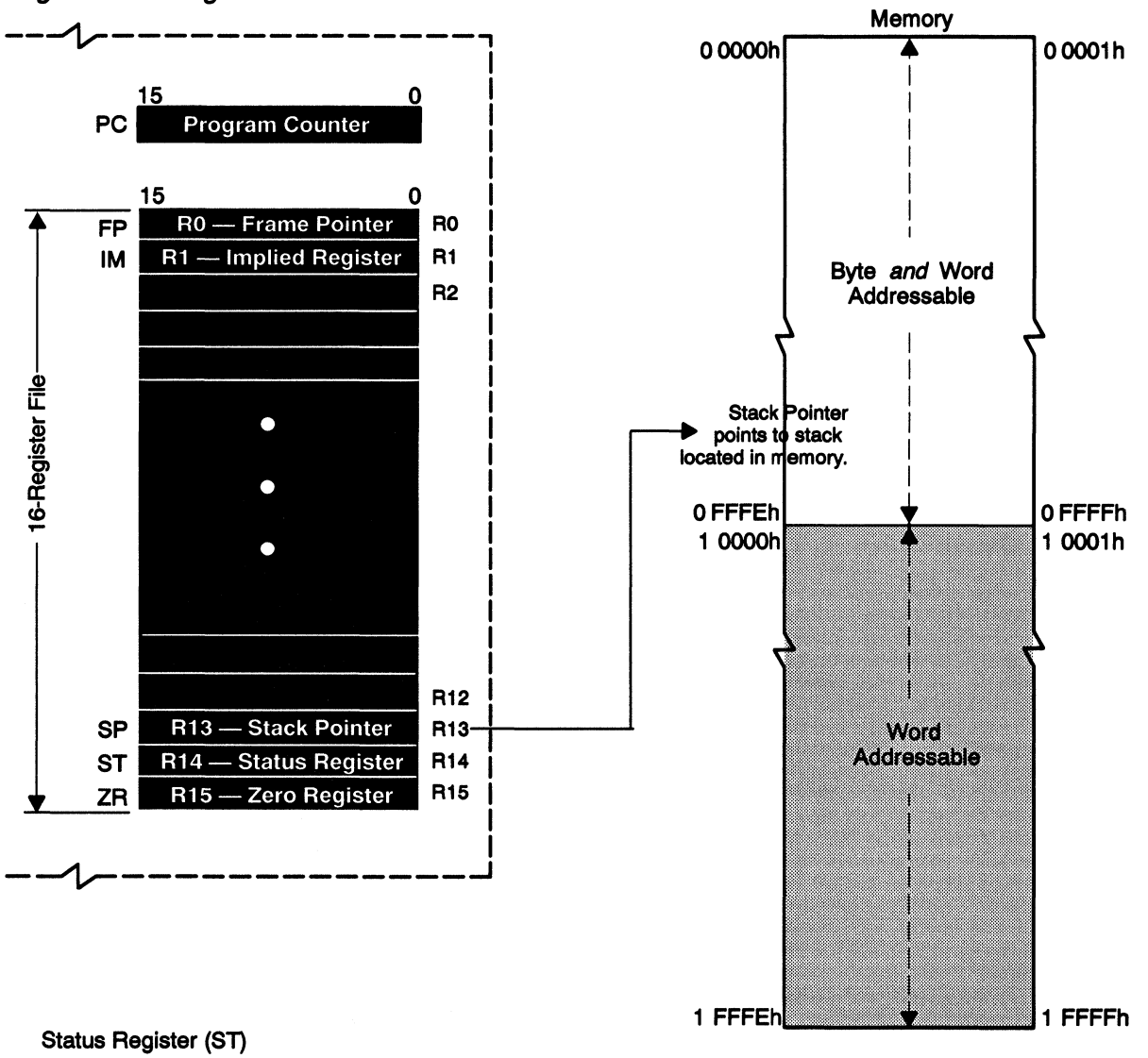
2.1 Programmer's Model

The TMS370C16 programmer's model consists of a 16-bit program counter and a 16-register file, which contains 11 general-purpose registers as well as the frame pointer, implied register, stack pointer, status register, and zero register. These are shown in Figure 2–1. The 'C16 may access RAM, EEPROM, EPROM, or ROM modules internally, depending on your device configuration. The 'C16 may also access the system module (further described in Section 3.1) that controls device operations such as stack location, reset, interrupts, I/O configurations, and the CLKOUT pin initialization. The 'C16 CPU and system module interface through the system address, data, and control buses to other modules such as the SPI, SCI, ADC, and gage drivers, depending upon your specific device configuration.

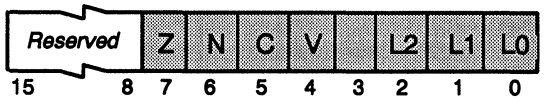
Figure 2–1 shows the register file and the memory accessible by the TMS370C16 CPU. The 16-register file is located in the CPU and includes five preassigned registers (R0, R1, R13, R14, and R15). This register file is discussed in further detail in Section 2.2, starting on page 2-4, and the status register (R14) and its bits, shown in the bottom of Figure 2–1, are described in more detail in subsection 2.2.4, page 2-6.

The program counter (PC), not part of the register file, contains the *word address* of an opcode or operand. The word address is applied to address lines A16–A1, with line A0 set to 0 (effectively multiplying the actual byte address by 2). This allows accessing data and executing code in a full 128K bytes of memory. The word address is further described in Section 2.3 on page 2-8, which includes a list of instructions using a 17-bit address (see Table 2–2 on page 2-9).

Figure 2–1. Programmer's Model



Status Register (ST)



The Status Register is covered in detail in subsection 2.2.4 on page 2-6.

Legend:

C = Carry N = Negative
V = Overflow Z = Zero
L2 – L0 = Interrupt priority level

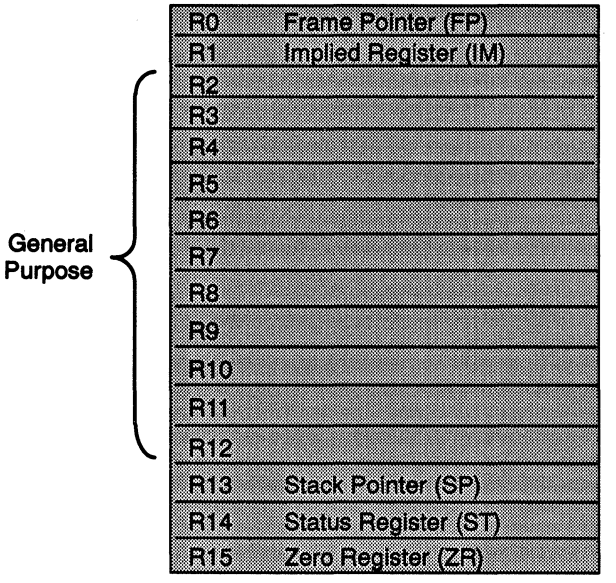
2.2 CPU Register File (R0 – R15)

The TMS370C16 CPU contains 16 registers, R0 – R15, that are not part of the memory map. Of the 16 registers, five can be used for the specialized functions listed in Figure 2–2 (registers R0, R1, R13, R14, and R15) or for general purposes.

R2 – R12, the 11 nonspecialized registers of the CPU register file, can be used for data manipulation for bit, byte (least significant byte), or word values. Take care when attempting to use any of the five specialized registers as general-purpose registers. The zero register (R15) reads as a zero value at all times, *and write values will be ignored.* Of the other specialized registers, R0 and R1 can be used conditionally, but R13 (stack pointer) and R14 (status) should not be used as general purpose at any time.

The values of the register file *are not* initialized by a reset. Your system software should initialize these registers during a startup procedure.

Figure 2–2. Registers R0 to R15



2.2.1 Frame Pointer, FP (R0)

The frame pointer can be used by high-level languages to allocate and deallocate procedure stack frames from the system stack. This register is implicitly used in the following instructions:

LINK	Link the FP to the current frame of the current SP (stack pointer) by pushing the FP onto the stack, setting the FP to the SP value, and then allocating designated words of stack.
UNLINK	Deallocate the current system stack frame by placing the FP contents in the SP and then retrieving the previous FP value from the system stack.
RTDU	Unlink and deallocate the current system stack frame by placing the FP value in the SP, retrieving the previous FP and PC contents from the stack (to return from a subroutine), and then subtracting a displacement from the SP.

2.2.2 Implied Register, IM (R1)

The implied register assists in dealing with 32-bit objects by serving as the most significant word of the two-word value. Also, in division operations, the IM holds the remainder.

The IM is used implicitly by the following instructions:

ASRL	Arithmetic shift right, longword (32-bit value)
ASR0L	Arithmetic shift right and round to 0, longword (32-bit value); add 1 if N[[ST]] and C[[ST]] are both 1
SHLL	Arithmetic shift left, longword (32-bit value)
DIVS	Division, signed (16- and 32-bit)
DIVU	Division, unsigned (16- and 32-bit)
EXTS	Sign-extend word to 32 bits
LSRL	Logically right-shift, longword (32-bit value)
MPYS	Signed word multiplication
MPYU	Unsigned word multiplication
TRUNCSL	Test to see if register can be truncated from 32 to 16 bits

2.2.3 Stack Pointer, SP (R13)

The stack pointer identifies the top of the stack — the location within the system stack to be used next (e.g., for storage of the current environment during interrupt processing). The stack also holds the return address for subroutine calls and provides a means of allocating procedure stack frames.

The SP is implicitly declared by the following instructions:

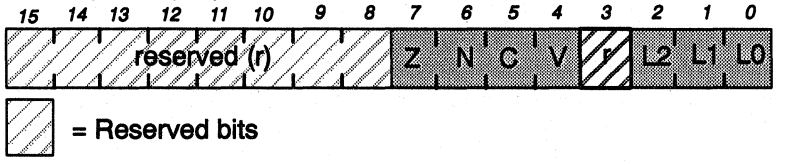
CALL	Jump to subroutine (return address on stack)
LINK	Link to current stack frame (FP to stack, SP to FP, and allocate requested words of space to the stack)
POP	Pull values from top of stack to register(s)
PUSH	Push values on top of stack from register(s)
RTDU	Unlink and deallocate current stack frame (return to former PC and new stack address)
RTI	Return from interrupt (retrieve PC and ST values from stack)
RTS	Return from subroutine (retrieve PC from stack)
TRAP	Generate one of 256 trap exceptions (push ST and PC + 1 onto stack, use vector offset and TRAP vector table to set PC, and set interrupt level at ST to all 1s)
UNLINK	Deallocate current stack frame (retrieve previous SP contents from FP register and retrieve old FP contents from stack)

Section 2.5 on page 2-11 contains a detailed discussion of the system stack.

2.2.4 Status Register, ST (R14)

The status register contains CPU status information from operations performed by the Arithmetic Logical Unit (ALU). The condition code bits Z (zero), N (negative), C (carry), and V (overflow) are typically altered during instruction execution. Status is based on the data object size — byte (8), word (16), or longword (32 bits) — of the just-executed instruction. The ST also contains the interrupt mask level bits L2 – L0.

Table 2–1. Status Register (ST) Bits



ST bit definitions:

Reserved (r): Bits reserved for future use. Data written to them are not retained.

Z: *Zero bit.* Set to 1 when an instruction generates a zero-value byte, word, or longword.

N: *Negative bit.* Generally set to the value of the most significant bit (e.g., sign bit) of an instruction's result. This is bit b7 for byte, b15 for word, and b31 for longword operations.

C: *Carry bit.* Set to 1 to indicate whether an unsigned overflow or underflow (carry/borrow) occurred during an arithmetic operation. Testing occurs as appropriate for the size of the data being operated on (byte, word, or longword). Some shift instructions use the C bit as a destination for the bit shifted. Bit load/store instructions treat the C bit as a bit accumulator.

V: *Overflow bit.* Generally set to 1 if a signed twos-complement overflow or underflow occurred during an arithmetic operation. Testing occurs as appropriate for the size of the data being operated on (byte, word, or longword).

L_n: *Interrupt-mask level bits (L2–L0).* Coded to specify interrupt levels of $000_2 - 111_2$ (0–7) with level **7 the highest** priority and level **0 the lowest**. Chapter 3 covers interrupt handling in detail (see Sections 3.7, 3.8, 3.9, and 3.10, beginning on page 3-19).

2.2.5 Zero Register, ZR (R15)

The zero register's contents are ***always* 0000h**. Thus, it is useful when a zero constant value is required.

This register can be used with indexed addressing (format **disp[Rn]*) to generate a direct address. When *Rn* is declared to be ZR (*disp[ZR]*), displacement *disp* becomes the operand's address (*disp* + 0). Thus, operands **disp[ZR]* and *&disp* are equivalent; use of the ampersand (&) operator for direct addressing is further explained in Section 4.4 on page 4-5.

Note: Register Considerations

1. Do not use R14 (status register) as a general-purpose register.
2. R15 (zero register) will always be read as a zero value; writing operations are ignored.

2.3 Program Counter (PC) and Address Bus

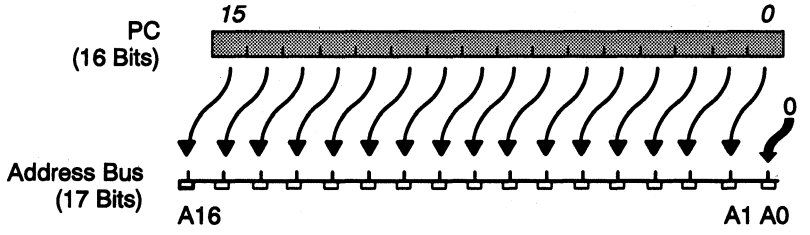
The PC is a 16-bit register, *not* included in the register file, that contains the *word* address of the instruction or instruction extension word that the CPU will fetch next. Because the PC uses the word-address data type, the instruction and the instruction extension words can be located at any **even** address in the entire 128K-byte memory address space of the 'C16. The term *word address* is defined in the note below.

Note: *Word Address* Definition

A **word address** is a 16-bit pointer that maps into a 128K-byte address space. Note that 17 bits are needed to fully address a 128K-byte space. Because the 'C16 requires that words begin on an even-byte boundary, the least significant bit of the word's address must be 0 with only the upper 16 bits of an address are required to access the word. A *word address* contains these 16 bits.

The PC holds the 16 *most significant bits* of the 17-bit memory address space. All instructions are word aligned; thus, *the least significant address bit* (bit 0) of all program references **always contains the value 0** (illustrated in Figure 2–3).

Figure 2–3. Program Counter to Address Bus Transition



Because of a pipeline architecture, the PC typically points to a memory address *two words beyond the currently executing instruction or to its extension word.* This relationship is graphically shown in Figure 2–4.

Figure 2–4. Relationship Between the PC and Memory Address

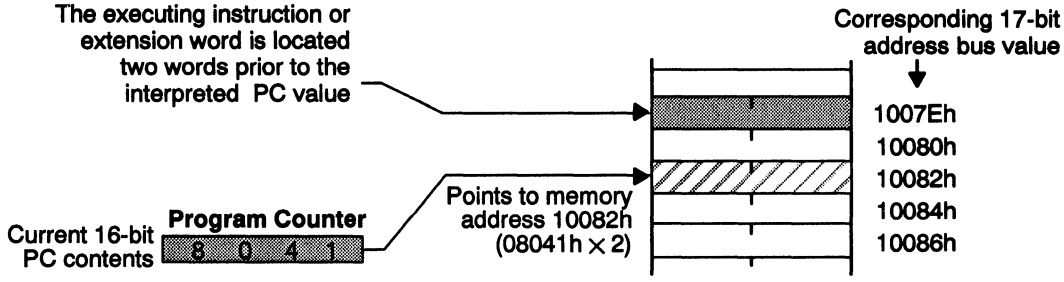


Figure 2–6 on page 2-13 describes execution flow during a jump to a subroutine. It also shows PC values and their corresponding address bus values. The note at the bottom of the figure explains the relationships.

The instructions in Table 2–2 use the PC register *(thus generating a 17-bit address).*

Table 2–2. Instructions That Use a 17-Bit Address

Instruction	Description
Bcond	Branch conditionally
BRBIT0	Branch if bit equals 0
BRBIT1	Branch if bit is a 1
CALL	Jump to (call) a subroutine (linkage provided)
DBNZ	Decrement register; branch only if result is 0
FMOV	Move (far) data to or from an address of up to 128K bytes
JMP	Jump unconditionally
RTDU	Return from subroutine and deallocate
RTI	Return from interrupt
RTS	Return from subroutine
TRAP	Generate one of 256 trap software interrupts; trap locations begin at address 08000h

The PC is also involved in the processing of reset, peripheral interrupts, and illegal opcode exceptions.

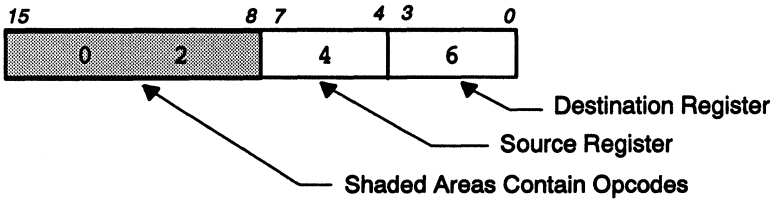
2.4 Instruction Organization

Bits are organized as shown in Figure 2–7. Instructions utilize one-, two-, or three-word formats as illustrated in Figure 2–5 for three different move instructions.

Figure 2–5. One-, Two-, and Three-Word Instruction Examples

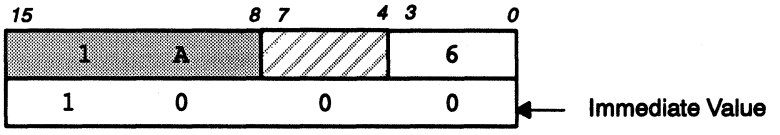
(a) One-Word Instruction

```
MOV  R4,R6          ; Move R4 to R6
```



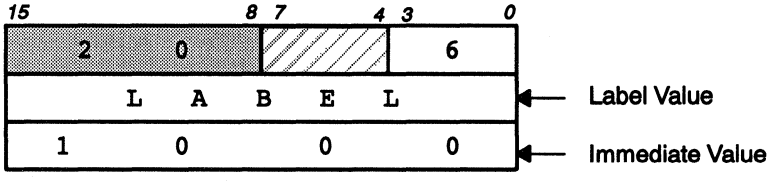
(b) Two-Word Instruction

```
MOV  #1000h,R6      ; Move 1000h to R6
```



(c) Three-Word Instruction

```
MOV #1000h,*LABEL[R6] ; Move 1000h to LABEL + R6
```



2.5 System Stack

The stack is a dedicated area of last-in/first-out RAM that is:

- ❑ Located in the first 64K bytes of memory
- ❑ Used for the storage of data that can describe an operating environment about to be exited or re-entered (such as the PC and ST values)
- ❑ Accessed by instructions that place data (PUSH instruction) into it from registers or retrieve data (POP instruction) from it into registers
- ❑ Used during a peripheral interrupt to store the operating environment that is to be exited (current ST and PC contents) before the address of the interrupt service routine is fetched
- ❑ Pointed to by the stack pointer (SP)

Table 2–3 lists instructions that use the stack:

Table 2–3. Instructions That Use The Stack

Instruction	Description	Detail
CALL	Jump to subroutine; provide return linkage	Push address of next instruction onto stack, then place destination value in PC (shown in Figure 2–6, page 2-13)
LINK	Link frame pointer (FP) to current stack; allocate stack space	Push FP onto stack, copy SP (old) to FP, then add displacement to SP for *new* SP value
POP	Copy stack words into specified registers	Specify range of registers affected
PUSH	Copy specified register words onto the stack	Specify range of registers affected
RTI	Return from interrupt	Pop PC and ST values from stack
RTS	Return from subroutine	Pop PC from stack (shown in Figure 2–6, page 2-13; RTS is at step 3 in the figure)
RTDU	Return from subroutine and deallocate current stack space	Can be a return from a CALL *but only if* subroutine executed a LINK instruction without an UNLINK instruction
TRAP	Generate one of 256 trap exceptions	Push ST and address of next instruction onto stack. Retrieve trap subroutine address from trap vector table and place in PC.
UNLINK	Unlink and deallocate stack frame	Place FP value in SP, then pop previous FP value from stack
ILLEGAL	Generate trap exception; this is caused when the instruction's illegal code of 0000h is decoded (one of several illegal opcodes that cause this)	Push ST and address of next instruction onto stack; place subroutine address from first trap location in PC

2.5.1 Stack Operation During Interrupts

A major use of the stack is to provide return linkage for a context switch. Steps of a typical context switch are as follows:

1) Context switch (e.g., interrupt) is recognized. Complete presently executing instruction.
2) Store present status register (ST) contents on the stack. Increment the stack pointer (SP) by two to the next memory address.
3) Store the present program counter value (PC) at the SP value (next address after the location where the ST is stored). Increment the SP by two.
4) Enter and execute the service routine for the context switch. When the routine is complete, reverse the process in steps 1 through 3 above to return to the environment present when the context switch was requested. This return is usually through an RTI (return from interrupt) instruction.
5) Decrement the SP by two. Retrieve the previous PC value at that address, and place it in the PC. Decrement the PC by two (this is explained in the RTI instruction description).
6) Decrement the SP by two. Retrieve the previous ST value at that address, and place it in the ST.

2.5.2 Stack Use with a Call

Figure 2–6 depicts how a stack is used when calling a subroutine with the CALL (jump to subroutine) instruction and then later returning to the calling environment. Numbered steps at the bottom of the figure correspond to circled numbers in the figure to explain execution sequence.

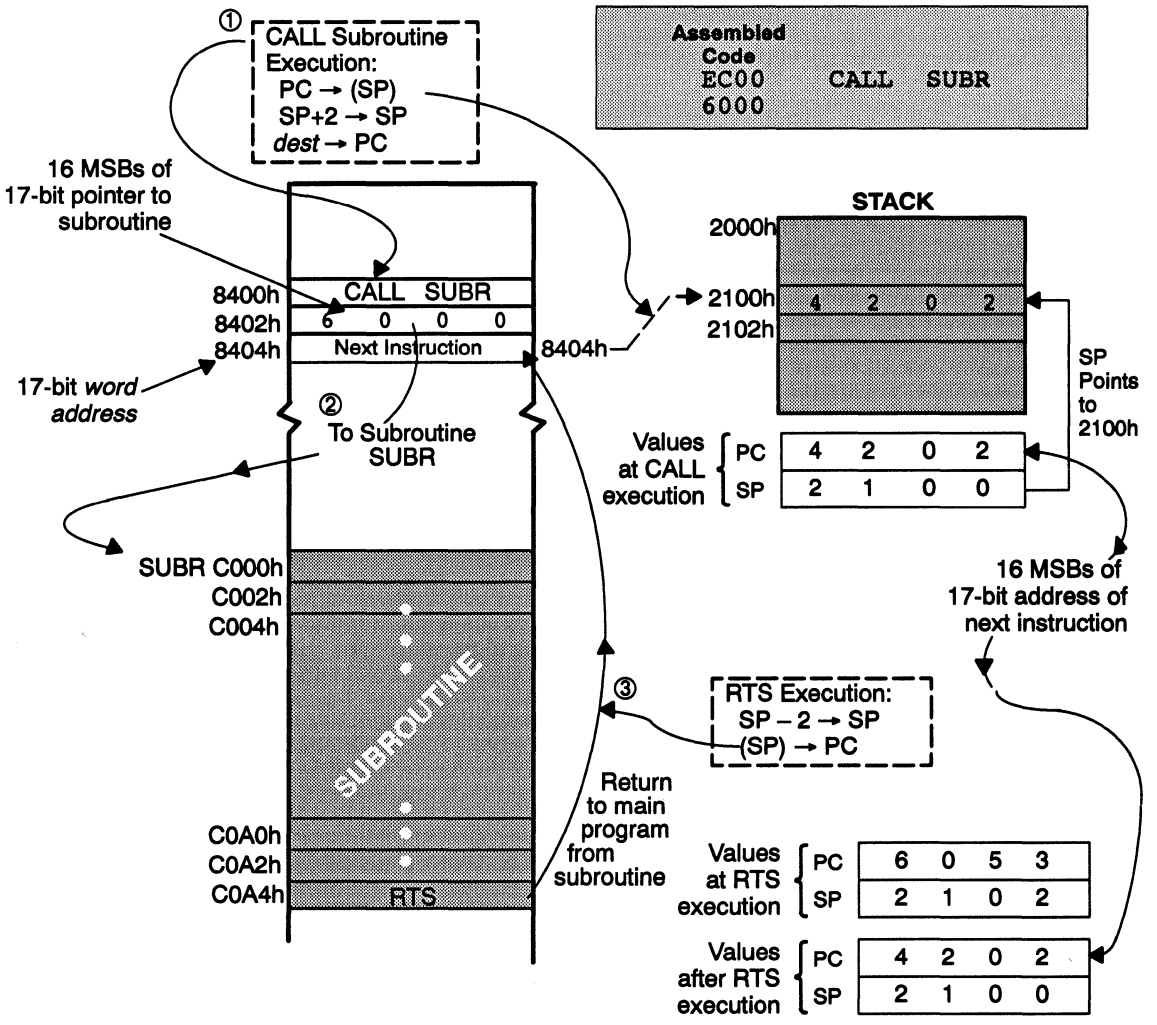
The stack increments by two after each *push* of a word value onto the stack. Conversely, the stack is decremented by two before each word is *pulled (popped)* from a stack.

Note: The SP Must Contain an *Even* Value

Make sure that the value stored in the SP (R13) is an *even* value (a 0 in address line A0). An odd value causes an illegal-access reset when the stack is addressed.

All implicit stack references by these instructions generate *word* read/write cycles to memory and thus are restricted to **even** addresses. The SP contents are used for address lines **A0 – A15**; thus, they should always be an even value. A nonaligned memory access generates a reset.

Figure 2–6. Example of Stack Use to and From a Subroutine



The CALL sequence:

① The CALL SUBR instruction causes a branch to subroutine SUBR with return values stored in the stack. Before the entry address of SUBR is placed in the PC:
   1) The present PC value (now pointing four bytes past the address containing the CALL opcode) is stored at the present contents in the stack pointer (SP).
   2) The SP is incremented by two.

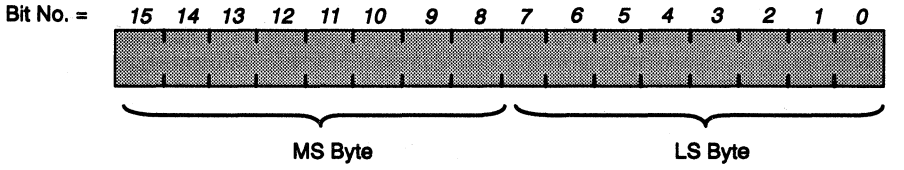
② The value of SUBR is placed in the PC. Execution begins at address C000h and continues-down to address C0A4h, which contains the last instruction in the subroutine — RTS (return from subroutine).

③ RTS returns the program back to the environment at the time of the CALL instruction by:
   1) Decrementing the SP by two to point to the address containing the PC value at the time of the CALL instruction.
   2) Placing the contents at the SP value into the PC. Execution begins at the next instruction after CALL.

2.6 Data Organization and Memory Mapping

Data resides in memory and on-chip registers with the most significant bit in the left-most position. Figure 2–7 shows the significance of bits and bytes.

Figure 2–7. Bit and Byte Numbering for Instructions, Registers, and Words



A **word** comprises two bytes:
- the *most significant* byte is on an ***even*** boundary, and
- the *least significant* byte occupies the next higher (*odd*) byte address.

Note: *Word Address* Definition

A **word address** is a 16-bit pointer that maps into a 128K-byte address space. Note that 17 bits are needed to fully address a 128K-byte space. Because the 'C16 requires that words begin on an even-byte boundary, the least significant bit of the word's address must be a 0; only the upper 16 bits of an address are required to access the word. A *word address* contains these 16 bits.

All *word* data in memory *must be aligned* on an *even* address.

For **byte** operations, the byte operand values are zero-extended to word length, are operated on as words, and produce a word result. *Register destinations* receive the entire word (the MSbyte zero-extended), but *memory destinations* receive only the LSbyte of the result. Thus, a byte moved to a register via the MOVB instruction zeroes the MSbyte of the register with the moved byte in the LSbyte. The same byte moved to a memory address affects only the destination byte addressed. This is illustrated in Figure 2–8 on page 2-16.

Figure 2–9 on page 2-17 shows how bits, bytes, and words are organized in memory and in the register file. Shown in the figure are the least and most significant bits and bytes. The accompanying explanations below the figure complete the description.

Figure 2–10 on page 2-18 shows a *typical* memory configuration and how the first and second 64K bytes of memory are divided into blocks for 1) **byte and word** access in the lower 64K bytes of memory and 2) **word-only** access in the higher 64K bytes.

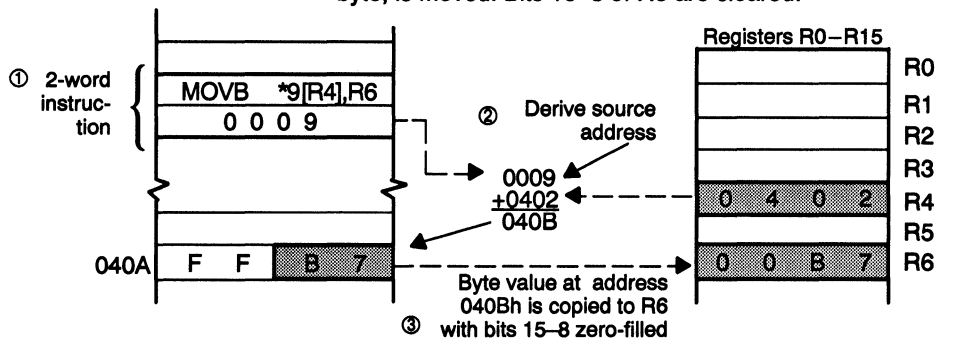
For purposes of this manual, these symbols have these meanings:

Symbol	Meaning	Example
(x)	Contents of register x or of memory at address x	(Rn) = the contents of Rn
((x))	Contents of memory designated by contents of x	(disp + (Rn)) = the contents within the value found by adding the contents of Rn with the displacement amount.

Figure 2–8. Differences in Memory and Register ByteDestinations

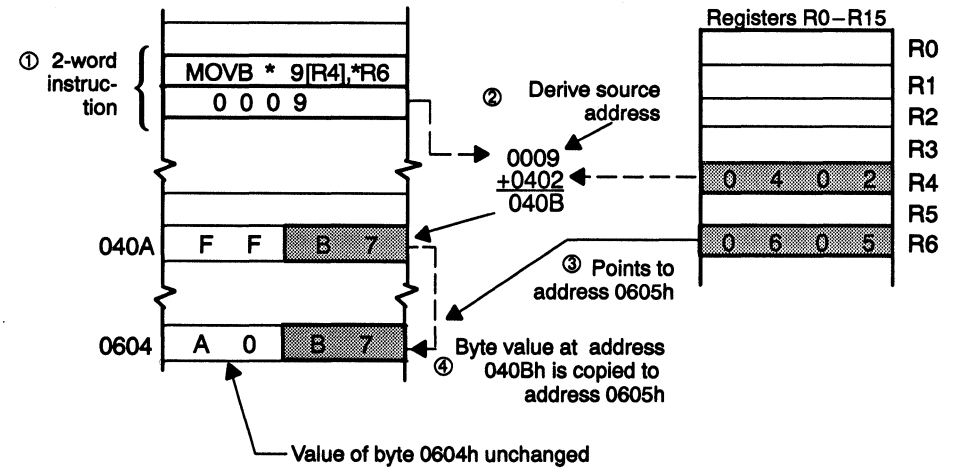
MOVB *9[R4],R6
Execution:
(0009h + (R4)) → (R6)

In this first example, a byte is moved to a register. The source value is found at the address derived by the sum of the 0009h displacement and the contents of R4, which contains 0402h. Thus, the value at address 040Bh, the least significant byte, is moved. Bits 15–8 of R6 are cleared.



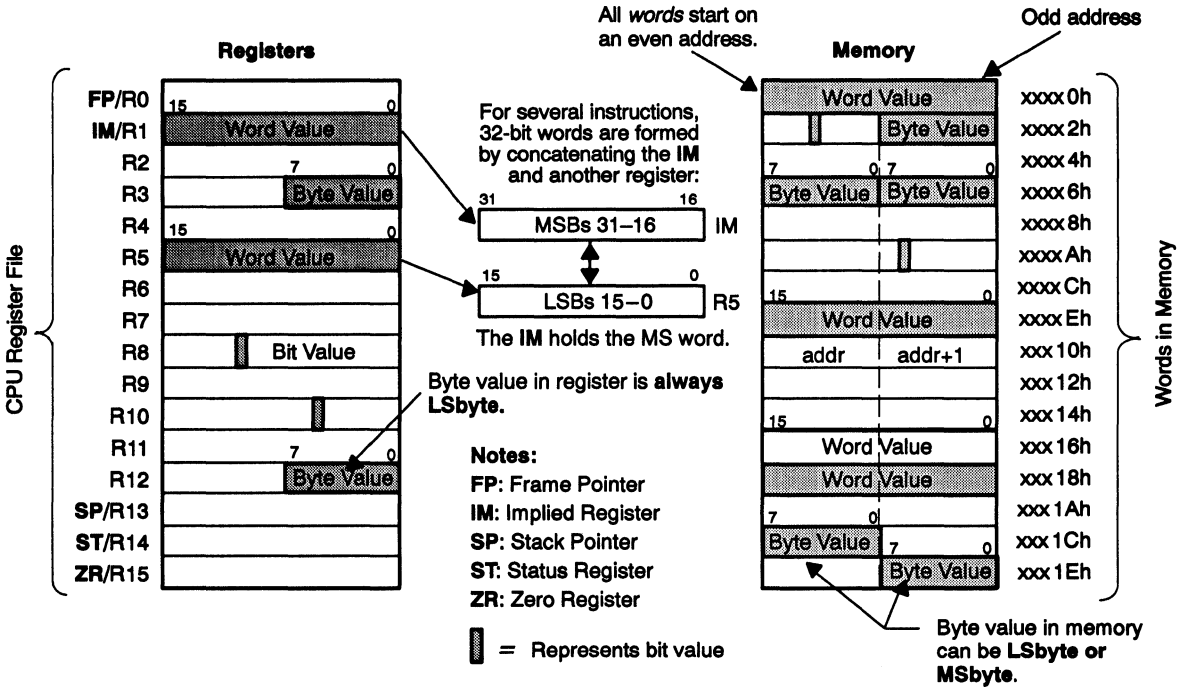
MOVB *9[R4],*R6
Execution:
(0009h + (R4)) → ((R6))

The above example is repeated, except that the destination is changed to a memory address with the destination register holding an *indirect address*. This example shows that the move affects only the designated byte in the destination, leaving the adjacent byte unchanged (no zero-filling occurs with a byte move to memory — *unlike* a byte move to a *register*).



Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

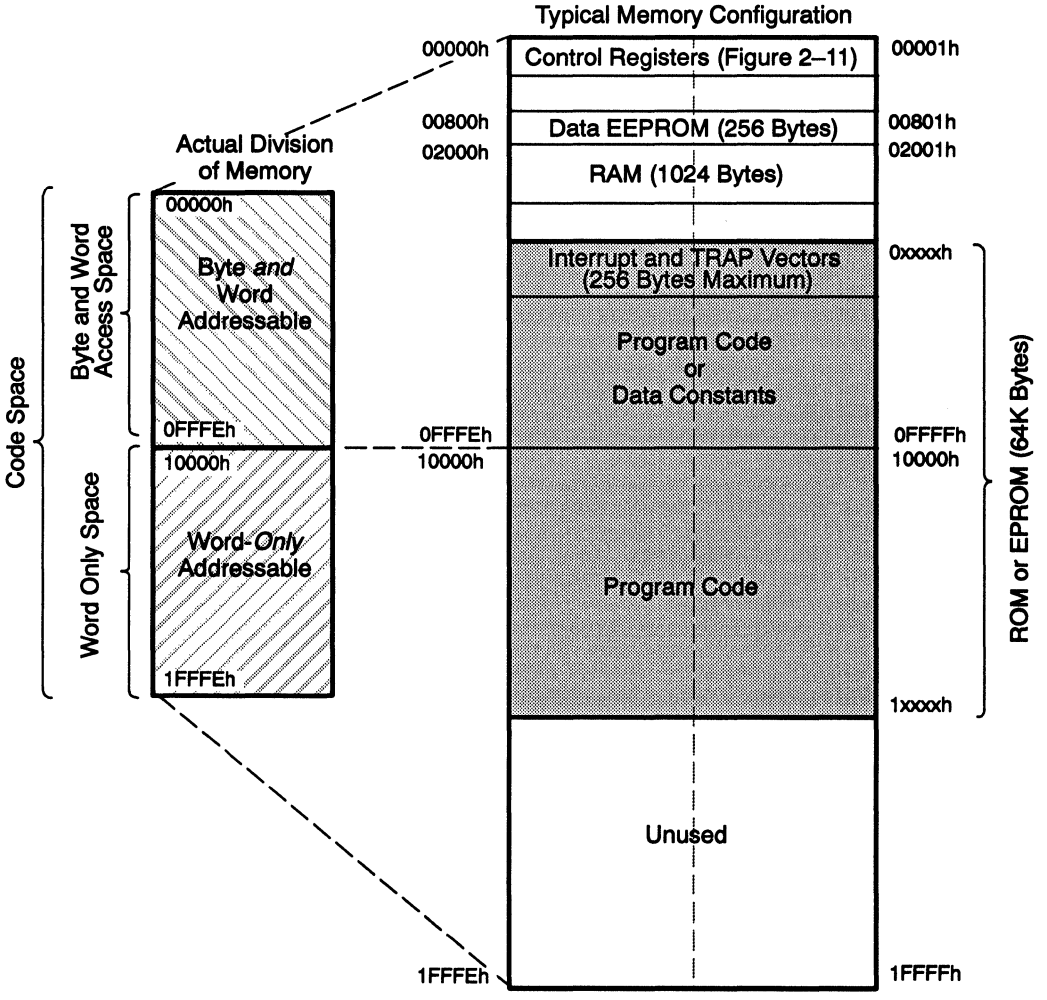
Figure 2–9. Data Organization Examples in Registers and Memory



Data restrictions depend upon their location in a register or in memory (as shown in Figure 2–9 above):

Data Size	Register	Memory
Single Bit	Can be any bit in any register in the register file.	Can be any bit in any byte in the first 64K bytes of memory.
Byte (8 bits)	The least significant byte of any register in the register file.	Can be any byte in the first 64K bytes of memory. Any adjacent byte is not affected.
Word (16 bits)	Can be any register in the register file.	Can be any byte pair where the most significant byte of the word is at an even address and the least significant byte is at the next higher byte address.
Long Word (32 bits)	Uses a register pair in the register file with the most significant word in the IM (implied register, R1).	Not applicable.

Figure 2–10. Typical 16-Bit Memory Map



As shown in Figure 2–10, two 64K-byte areas concatenate to form 128K bytes of addressable memory. The generic view on the left shows that the lower-address half can be accessed as either byte or word, and the higher-address half is accessible as word-only by such instructions as FMOV and CALL. The right side of the figure is an *example* of possible code and data utilization. The actual size of the memory module is device specific. See your specific device data sheet to determine the size of the memory modules for your particular device. The lowest memory addresses contain the control registers, which are expanded in Figure 2–11 (next page).

Figure 2–11. Location and Names of Control Registers

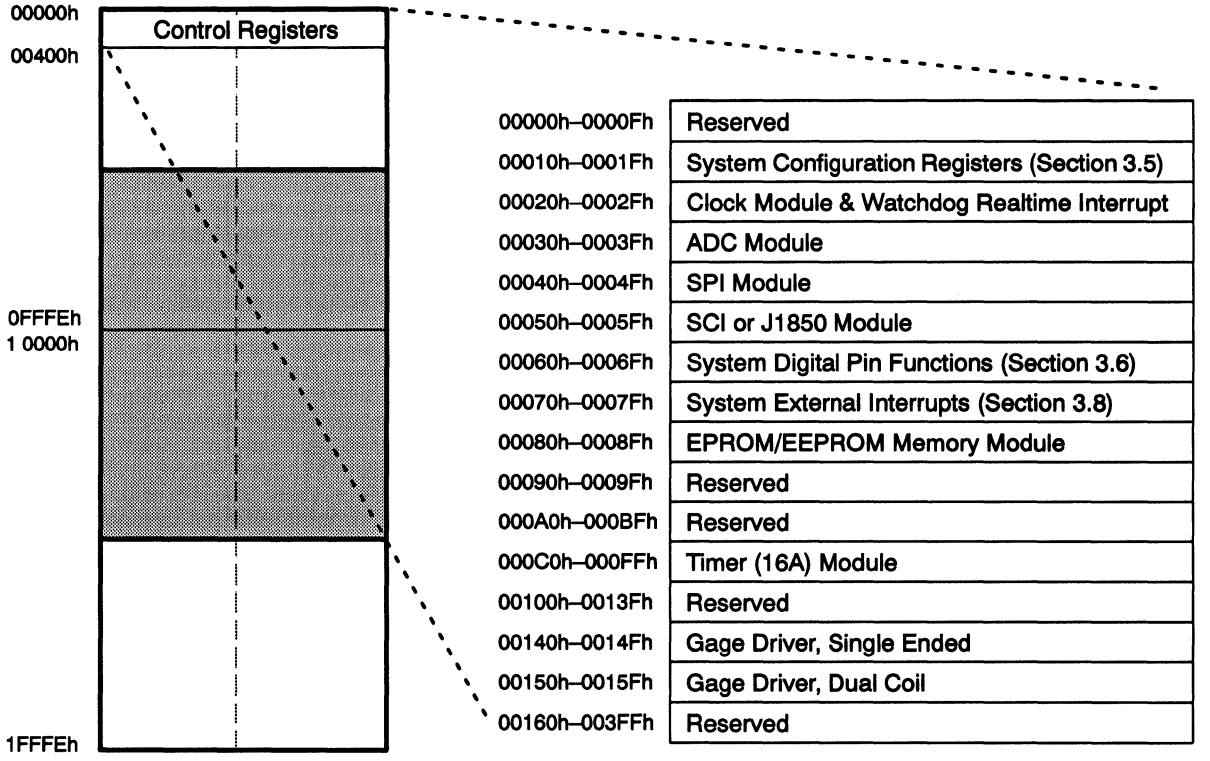


Figure 2–11 lists the 16 control-register groups in the lowest 1K bytes of memory. Each register group is 16 bytes and contains the working registers for each module or for the system configuration. These registers are further described in Section 3.5 on page 3-7.

Chapter 3

TMS370C16 System Configuration

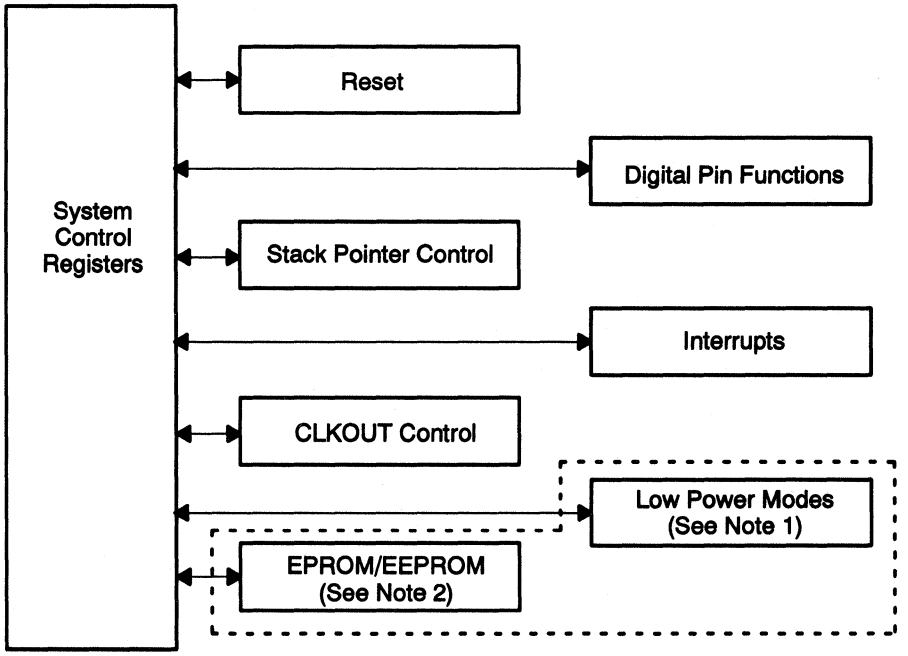
This chapter discusses system configuration requirements, I/O, interrupts, reset, and low-power modes of the TMS370C16 CPU. Features and options are described, including the registers that control the configuration. This chapter covers the following topics:

Topic **Page**

3.1 System Configuration Overview

The system module controls device operations such as clock source, stack location, reset, interrupts, and I/O. The actual number of external interrupts and I/O pins is device specific; consult the data sheet for a particular device. Certain device status information is also contained within the system module. The system module block diagram is shown in Figure 3–1.

Figure 3–1. System Block Diagram



Notes: 1. See the *Clock Modules Reference Guide.*
2. See the *EEPROM/EPROM Modules Reference Guide.*

3.2 System Reset Operation

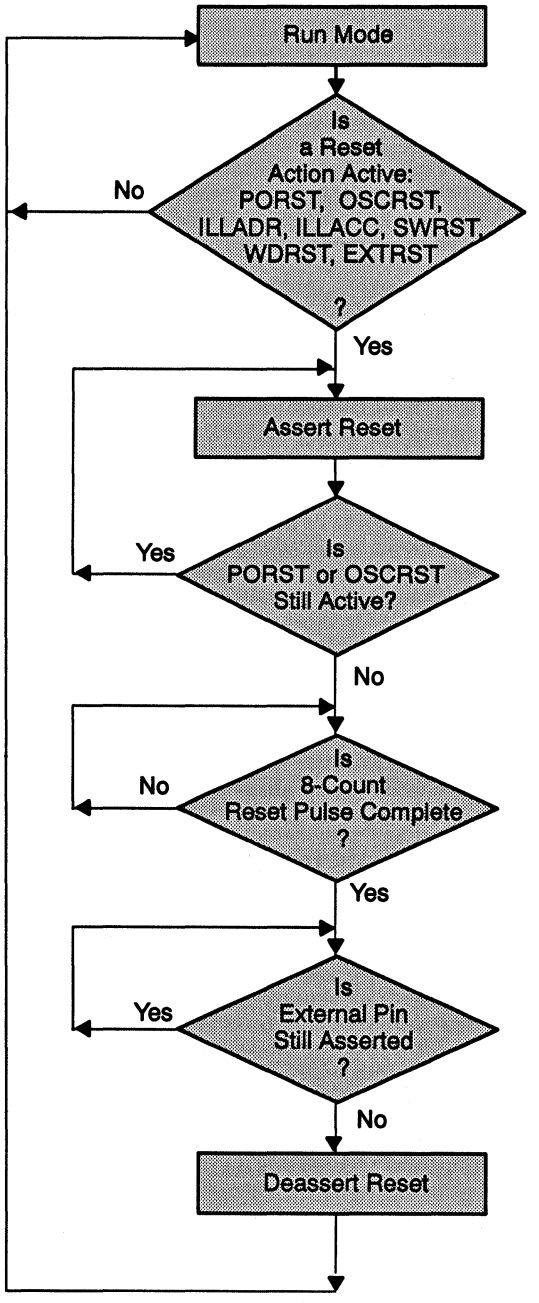
The system reset operation ensures an orderly start-up sequence for the TMS370C16 CPU-based device. Seven actions can cause a system reset to the device; six of these are internally generated, while the $\overline{\text{RESET}}$-pin interrupt is controlled externally.

- $\overline{\text{RESET}}$ Pin. A negative edge can trigger a signal on this external pin.
- Watchdog (WD) Timer Overflow. A watchdog-generated reset occurs if the WD timer overflows or an improper value is written to either the WD key register or the WD control register. (See your *Watchdog Timer and Real-Time Interrupt Reference Guide* for details on these registers.)
- Software-Generated Reset. Writing a 0 to the RESET0 bit (SCR0.6) or a 1 to the RESET1 bit (SCR0.7) causes a reset (SCR0 is the system control register 0, as shown in Figure 3–3 on page 3-7.)
- Illegal Address Access. Attempting to access a nonmemory (not implemented) address causes a reset. (This action is device specific, relative to the memory configuration.)
- Oscillator Reset. Operation of the oscillator outside of the recommended operating range, as indicated by the OSCRST bit of the system reset status register (subsection 3.5.3, page 3-10), causes the clock module to issue a reset. See the *Clock Modules Reference Guide* for more information.
- V_{CC} Out-of-Range. Operation with V_{CC} outside of the recommended operating range may also act as a brownout indicator in addition to ensuring proper operation on power-up sequences.
- Illegal Access. Attempting to access a word by using an odd address causes a reset.

Once a reset source is activated, the external $\overline{\text{RESET}}$ pin is driven active low for a minimum of eight SYSCLK cycles. This allows the 'C16 CPU-based device to reset any external devices connected to the $\overline{\text{RESET}}$ pin. Normally, the reset logic holds the 'C16 device in a reset state for eight SYSCLK cycles; however, if a V_{CC} out-of-range condition or oscillator failure occurs (or the $\overline{\text{RESET}}$ external pin is held low), then the reset logic holds the device in a reset state for as long as these conditions exist.

Figure 3–2 shows the reset state diagram for the 'C16 device in the normal run mode.

Figure 3–2. Reset State Diagram — Normal Run Mode



Note: Reset actions are indicated in the system reset status register described on page 3-10.

After a reset, the program determines the source of the reset by reading the contents of the system reset status register (SRSR, shown in Figure 3–3 on page 3-7). There is one status bit for each of the seven sources that can cause a reset.

Once a reset is activated, the following sequence of events occurs in the 'C16:

1) The CPU registers and module control registers are initialized to their reset state. The ST interrupt mask bits are set to all 1s to prevent any interrupt request, including nonmaskable interrupts (NMIs).

2) The correct index value to the trap table base address is computed.

3) The service-routine address is read from address 8002h.

4) The prefetch pipeline is reloaded.

The reset sequence takes six cycles from the time the reset is released until the first opcode fetch begins. During a reset, RAM contents remain unchanged, and the module control register bits are initialized to their reset state.

To generate an external reset pulse on the $\overline{\text{RESET}}$ pin, a low-level pulse duration of as little as a few nanoseconds is usually effective; however, pulses of one SYSCLK cycle are recommended to guarantee that the device acknowledges the reset. A typical reset circuit required for the 'C16 CPU-based device consists of a 10-kilohm pullup resistor from the $\overline{\text{RESET}}$ pin to V_{CC}. Only this single resistor is needed if a primary voltage regulator or brownout detection circuit is on your device. See the specific device data sheet to determine whether additional circuitry is required.

3.3 CLKOUT Pin Function Selection

You can select the CLKOUT pin to operate as one of four different functions:

- ❑ Digital I/O
- ❑ Watchdog clock (WDCLK) output
- ❑ External clock (ECLK) output
- ❑ System clock (SYSCLK) output

The function is determined by two clock source control bits, CLKSRC1 and CLKSRC0 (SCR1.7 and SCR1.6 respectively, shown in Figure 3–3 on page 3-7). Table 3–1 illustrates the CLKOUT pin function selection options.

Table 3–1. CLKOUT Pin Function Options

	CLKSRC1	CLKSRC0
Digital I/O	0	0
WDCLK	0	1
ECLK	1	0
SYSCLK	1	1

For more information, see subsection 3.5.2 on system control register 1 on page 3-9, the specific device data sheet, or the *Clock Modules Reference Guide.*

3.4 Parallel Signature Analysis Operation (CRC Generator)

The TMS370C16 device contains an internal 16-bit parallel signature analysis (PSA) circuit that provides a continuous cyclic redundancy check (CRC) function. Two associated registers, PSAR1 and PSAR2 (located at addresses 0001Eh and 0001Fh in the system configuration register), determine a unique 16-bit signature. (The system configuration register is further described in Section 3.5 and in Figure 3–3 on the next page.)

When any memory location (RAM, EEPROM, ROM, EPROM, or control register) is read, the contents of the PSA registers are updated (register bits are described in subsection 3.5.5 on page 3-13). You can create a predetermined signature by initializing the PSA registers to a known value and then reading all memory locations. It is suggested that you read both PSA registers as a single word (*avoid multiple reads such as reading each byte individually*).

3.5 System Configuration Registers

The TMS370C16 system configuration registers are shown in Figure 3–3 and are discussed in detail in the following sections. These registers can be accessed in either byte or word mode.

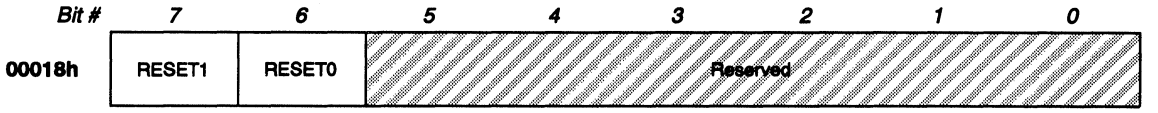
Figure 3–3. System Configuration Registers

Address	Register Mnemonic	7	6	5	4	3	2	1	0	Register Name
00010h	—	Reserved								—
00011h	—	Reserved								—
00012h	—	Reserved								—
00013h	—	Reserved								—
00014h	—	Reserved								—
00015h	—	Reserved								—
00016h	—	Reserved								—
00017h	—	Reserved								—
00018h	SCR0	RESET1	RESET0	Reserved						System Control Register 0
00019h	SCR1	CLKSRC1	CLKSRC0	Reserved		VCCAON	Reserved			System Control Register 1
0001Ah	SRSR	PORST	OSCRST	Reserved	ILLADR	ILLACC	SWRST	WDRST	EXTRST	System Reset Status Register
0001Bh	SSR	Reserved		HPO	Rserved	VCCAOR	Reserved			System Status Register
0001Ch	—	Reserved								—
0001Dh	—	Reserved								—
0001Eh	PSAR1	PSA15	PSA14	PSA13	PSA12	PSA11	PSA10	PSA9	PSA8	Parallel Signature Analysis Reg. 1
0001Fh	PSAR2	PSA7	PSA6	PSA5	PSA4	PSA3	PSA2	PSA1	PSA0	Parallel Signature Analysis Reg. 2

3.5.1 System Control Register 0 (SCR0)

The system control register 0 (SCR0) controls the software reset capability of TMS370C16 CPU-based devices.

System Control Register 0 (SCR0)



R = Read, *W* = Write, *–n* = value after reset (0, 1, x = indeterminate)

Bits 7 & 6 **RESET1/RESET0.** Software Reset.
These bits, which control the software reset function of the device, must be written to at the same time. Writing a 1 to RESET1 *or* a 0 to RESET0 causes a global reset to occur as shown in the following table:

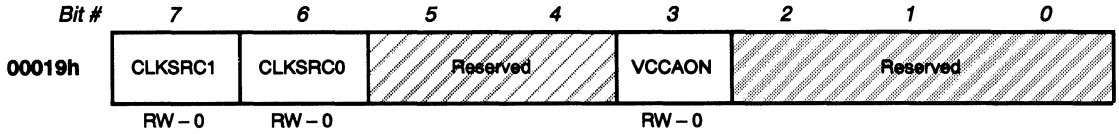
RESET1	RESET0	Resulting Action
0	0	Global reset
0	1	—
1	0	Global reset
1	1	Global reset

Bits 5 – 0 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

3.5.2 System Control Register 1 (SCR1)

The system control register 1 (SCR1) controls the CLKOUT pin function and the analog power supply enable.

System Control Register 1 (SCR1)



R = Read, *W* = Write, *–n* = value after reset (0, 1, x = indeterminate)

Bits 7 & 6 **CLKSRC1–0.** Clockout Pin Source Select.
These bits control the selection of the CLKOUT pin function.

CLKSRC1	CLKSRC0	CLOCKOUT Pin Function
0	0	Digital I/O mode
0	1	WDCLK clock output mode (watchdog clock)
1	0	ECLK output mode (external clock)
1	1	System Clock (SYSCLK) output mode

Bits 5 & 4 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

Bit 3 **VCCAON.** V_{CCA} (Analog Power Supply) Enable.
This bit controls the ability of the primary voltage regulator or the brown-out detect circuit to turn the analog power supply (V_{CCA}) on and off.
0 = Analog power supply is disabled.
1 = Analog power supply is enabled.

Bits 2–0 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

3.5.3 System Reset Status Register (SRSR)

The system reset status register (SRSR) contains system-reset history status information. These bits should be cleared after being read.

System Reset Status Register (SRSR)

Bit #	7	6	5	4	3	2	1	0
0001Ah	PORST	OSCRST	Reserved	ILLADR	ILLACC	SWRST	WDRST	EXTRST
	RC–x	RC–x		RC–x	RC–x	RC–x	RC–x	RC–x

R = Read, *C* = Clear only, *–n* = Value after reset (0, 1, x = Indeterminate)

Bit 7 **PORST.** Power On Reset Status.
This bit indicates the status of digital power to the chip. When a reset occurs because V_{CCD} is out of regulation, bit 7— PORST— is set. Reset is active while V_{CCD} is out of regulation, and for eight cycles afterward. When this bit is set to a 1, all other bits are indeterminate.

0 = No reset. V_{CCD} is not out of regulation.
1 = Reset because V_{CCD} out of regulation.

Bit 6 **OSCRST.** Oscillator Reset Status.
Reset occurred because of an oscillator fail condition. Ignore this bit if there is no phase lock loop oscillator.

0 = No oscillator fail conditions.
1 = Reset due to oscillator fail condition.

Bit 5 **Reserved.**
Writing to this bit has no effect, and reads are undefined.

Bit 4 **ILLADR.** Illegal Address Reset Status.
This reset occurs when an unimplemented memory address is accessed.

0 = No illegal address conditions.
1 = Reset due to illegal address access.

Bit 3 **ILLACC.** Illegal Access Reset Status.
This reset occurs when a word access occurs on a byte (odd address value) boundary.

0 = No illegal access conditions
1 = Reset due to illegal access.

Bit 2 **SWRST.** Software Reset Status.
This reset occurs when a #1 is written to bit SCR0.7 or a 0 is written to bit SCR0.6.

0 = No reset.
1 = Software reset occurred.

Bit 1 **WDRST.** Watchdog Reset Status.
See your *Watchdog Timer and Real-Time Interrupt Module Reference Guide* to determine whether this bit applies to your device.

0 = No reset.
1 = Reset due to watchdog timer overflow.

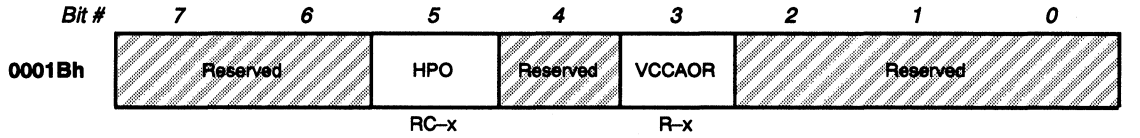
Bit 0 **EXTRST.** External Reset Status.

0 = No reset.
1 = This bit is set when the external $\overline{\text{RESET}}$ pin is pulled low by any source, including an internal reset.

3.5.4 System Status Register (SSR)

The system status register (SSR) contains status information about the operational modes of the device.

System Status Register (SSR)



R = Read, *C* = Clear only, *–n* = Value after reset (0, 1, x = Indeterminate)

Bits 7 & 6 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

Bit 5 **HPO.** Hardware Protect Override.
The hardware protect override function allows protected EEPROM bits to be written to and enables EPROM programming. To set this bit, external pin INT1 must be at 12 V on the rising edge of $\overline{\text{RESET}}$. If INT1 is less than 12 V, the bit is a 0. You can disable this function by writing a 0 to it.

0 = Normal mode.
1 = HPO mode.

Bits 4 **Reserved.**
Writing to this bit has no effect, and reads are undefined.

Bit 3 **VCCAOR.** V_{CCA} (Analog Power Supply) Out of Regulation.
This bit shows the status of the internal V_{CCA} signal.

0 = V_{CCA} is within regulated range.
1 = V_{CCA} is out of regulated range.

Bits 2–0 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

3.5.5 Parallel Signature Analysis Registers (PSARn)

The parallel signature analysis register 1 (PSAR1) contains the MSbyte of the PSA, and the parallel signature analysis register 2 (PSAR2) contains the LSbyte of the PSA.

Parallel Signature Analysis Register 1 (PSAR1)

Bit #	7	6	5	4	3	2	1	0
0001Eh	PSA15	PSA14	PSA13	PSA12	PSA11	PSA10	PSA9	PSA8
	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0

R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (*0, 1,* or *x* = indeterminate)

Bits 7 – 0 **PSA15–PSA8.** Parallel Signature Analysis Data Bits 15 – 8. The value read from this register is the MSbyte of the most recent PSA routine.

Parallel Signature Analysis Register 2 (PSAR2)

Bit #	7	6	5	4	3	2	1	0
0001Fh	PSA7	PSA6	PSA5	PSA4	PSA3	PSA2	PSA1	PSA0
	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0

R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (*0, 1,* or *x* = indeterminate)

Bits 7 – 0 **PSA7–PSA0.** Parallel Signature Analysis Data Bits 7 – 0. The value read from this register is the LSbyte of the most recent PSA routine.

3.6 General-Purpose Digital Pin Functions

Device pins can be configured for general-purpose digital pin functions *except* for those pins:

- That are device operation pins (V_{CC}, V_{SS}, $\overline{RESET}$, INT1, etc.).
- That are *required* for module-specific operation (for the SPI, ADC, gage drivers, etc.)

The total number of digital pins available is device specific. Refer to the specific device data sheet to determine the exact number of digital pins available, pin locations, naming conventions, and control registers. This section describes the different types of digital pin functions available and how they are controlled.

The digital I/O control and status register (Figure 3–4) allows a maximum of 32 output/control functions, 32 input/status functions, and 32 bidirectional I/O pin functions. The output pin functions are also referred to as *control* pins — they can be used to turn particular internal modules on or off and are not actually tied to an external pin. The input pin functions are also referred to as *status* pins because they can be used to determine the status of internal signals on the device as well as to serve as general-purpose input pins. For example, you could use these configurations to tie an input/status function to the low-side driver over-current detection circuitry, or to tie an output/control function internally to the V_{CCA} analog voltage output to control the primary voltage regulator during on and off V_{CCA}.

The control registers for digital I/O (DIO) pins are located at addresses 0060h to 006Fh and are shown in Figure 3–4.

Address	Ports	Functions
0060h – 0063h	Output 1, 2, 3, 4	Output/control only. Pins for output/control ports 1, 2, 3, and 4
0064h – 0067h	Input 1, 2, 3, 4	Input/status only. Pins for Input/status ports 1, 2, 3, and 4
0068h – 006Fh	I/O A, B, C, D	Pins for I/O ports A, B, C, and D, with each port using one byte for I/O configuration and one byte for pin value.

The following sections explain the operation of the DIO control registers. The number of DIO control registers available depends on the 'C16 device. Usually, all digital pins available are configured as bidirectional I/O pins, and not output or input only. This configuration selection is determined during the manufacture cycle and cannot be changed by software. See the specific device data sheet for more information.

Figure 3–4. Digital I/O Control and Status Registers

Addr	Reg Mnem	7	6	5	4	3	2	1	0	Reg Name
00060h	OCR1	1OUT7	1OUT6	1OUT5	1OUT4	1OUT3	1OUT2	1OUT1	1OUT0	Output/ Control Register 1
00061h	OCR2	2OUT7	2OUT6	2OUT5	2OUT4	2OUT3	2OUT2	2OUT1	2OUT0	Output/ Control Register 2
00062h	OCR3	3OUT7	3OUT6	3OUT5	3OUT4	3OUT3	3OUT2	3OUT1	3OUT0	Output/ Control Register 3
00063h	OCR4	4OUT7	4OUT6	4OUT5	4OUT4	4OUT3	4OUT2	4OUT1	4OUT0	Output/ Control Register 4
00064h	ISR1	1INS7	1INS6	1INS5	1INS4	1INS3	1INS2	1INS1	1INS0	Input/ Status Register 1
00065h	ISR2	2INS7	2INS6	2INS5	2INS4	2INS3	2INS2	2INS1	2INS0	Input/ Status Register 2
00066h	ISR3	3INS7	3INS6	3INS5	3INS4	3INS3	3INS2	3INS1	3INS0	Input/ Status Register 3
00067h	ISR4	4INS7	4INS6	4INS5	4INS4	4INS3	4INS2	4INS1	4INS0	Input/ Status Register 4
00068h	ADIR	ADIR7	ADIR6	ADIR5	ADIR4	ADIR3	ADIR2	ADIR1	ADIR0	I/O Port A Direction Register
00069h	ADATA	ADATA7	ADATA6	ADATA5	ADATA4	ADATA3	ADATA2	ADATA1	ADATA0	I/O Port A Data Register
0006Ah	BDIR	BDIR7	BDIR6	BDIR5	BDIR4	BDIR3	BDIR2	BDIR1	BDIR0	I/O Port B Direction Register
0006Bh	BDATA	BDATA7	BDATA6	BDATA5	BDATA4	BDATA3	BDATA2	BDATA1	BDATA0	I/O Port B Data Register
0006Ch	CDIR	CDIR7	CDIR6	CDIR5	CDIR4	CDIR3	CDIR2	CDIR1	CDIR0	I/O Port C Direction Register
0006Dh	CDATA	CDATA7	CDATA6	CDATA5	CDATA4	CDATA3	CDATA2	CDATA1	CDATA0	I/O Port C Data Register
0006Eh	DDIR	DDIR7	DDIR6	DDIR5	DDIR4	DDIR3	DDIR2	DDIR1	DDIR0	I/O Port D Direction Register
0006Fh	DDATA	DDATA7	DDATA6	DDATA5	DDATA4	DDATA3	DDATA2	DDATA1	DDATA0	I/O Port D Data Register

Note: See the specific device data sheet for the actual digital pin implementation.

3.6.1 Digital Output/Control Registers (OCR*n*)

Writing to bit(s) in the digital output/control registers (OCR1, OCR2, OCR3, and OCR4) outputs values to the bit's corresponding function(s) — such as communication to an internal module or an external pin. OCR1 is illustrated below. OCR2. OCR3, and OCR4 operate identically to OCR1 but are not shown.

Digital Output/Control Register 1 (OCR1)

Bit #	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
00060h	1OUT7	1OUT6	1OUT5	1OUT4	1OUT3	1OUT2	1OUT1	1OUT0
	RW–0	*RW–0*	*RW–0*	*RW–0*	*RW–0*	*RW–0*	*RW–0*	*RW–0*

R = Read, *W* = Write, *–n* = Value after reset (0, 1, x = indeterminate)

Bits 7–0 **1OUT7 – 1OUT0.** Digital outputs to corresponding functions. The values written to any of selected bit(s) 1OUT7 to 1OUT0 control the state output of the corresponding function(s).

0 = Output a 0 (V_{OL}) value to the selected function.
1 = Output a 1 (V_{OH}) value to the selected function.

3.6.2 Digital Input/Status Registers (ISR*n*)

Reading a bit in one of the four digital input/status registers (ISR1, ISR2, ISR3, and ISR4) reads the bit value at the corresponding input function. Functions could be values from such points as a module flag, external pin, etc. ISR1 is illustrated below. ISR2, ISR3, and ISR4 operate identically to ISR1 but are not shown.

Digital Input/Status Register 1 (ISR1)

Bit #	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
00064h	1INS7	1INS6	1INS5	1INS4	1INS3	1INS2	1INS1	1INS0
	R–x	*R–x*	*R–x*	*R–x*	*R–x*	*R–x*	*R–x*	*R–x*

R = Read, *W* = Write, *–n* = Value after reset (0, 1, x = indeterminate)

Bits 7–0 **1INS7–1INS0.** Digital input/status at corresponding functions. The values read at any selected bit(s) 1INS7–1INS0 show values at their corresponding functions:

0 = Read V_{IL} on the corresponding function.
1 = Read V_{IH} on the corresponding function.

3.6.3 Digital Port Direction and Port Data Registers (*x*DIR and xDATA)

The TMS370C16 CPU has four digital ports — A, B, C, and D. Each port has a pair of registers that work together. The direction register for the port designates each bit in the corresponding data register as either an input or output.

- The **port direction register** (ADIR, BDIR, CDIR, and DDIR for ports A to D respectively) bit values designate a corresponding pin in the data register as an input (clear bit to 0) or an output (set bit to 1).
- The **port data register** (ADATA, BDATA, CDATA, and DDATA for ports A to D respectively) bits can be read from or written to, depending upon their status as set in the port direction register.

For example, to read bit A7, clear bit ADIR7 to 0 (to become an input); then read bit ADATA7. To write to A7, set ADIR7 to 1 (becomes an output) and write a value to ADATA7. This applies to the other ports also (BDIR/BDATA, CDIR/CDATA, and DDIR/DDATA).

Registers ADIR and ADATA are shown on the next page. The combinations of BDIR/BDATA, CDIR/CDATA, and DDIR/DDATA operate identically to ADIR/ADATA but are not shown.

I/O Port A Direction Register (ADIR)

Bit #	7	6	5	4	3	2	1	0
0006Ah	ADIR7	ADIR6	ADIR5	ADIR4	ADIR3	ADIR2	ADIR1	ADIR0
	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0

R = Read, *W* = Write, *–n* = Value after reset (0, 1, x = indeterminate)

Bits 7–0 **ADIR7–ADIR0.** Control direction of pins A7–A0.
The value written to any one of these bits controls the direction of this bidirectional pin.
0 = The pin is an input.
1 = The pin is an output.

I/O Port A Data Register (ADATA)

Bit #	7	6	5	4	3	2	1	0
0006Bh	ADATA7	ADATA6	ADATA5	ADATA4	ADATA3	ADATA2	ADATA1	ADATA0
	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0

R = Read, *W* = Write, *–n* = Value after reset (0, 1, x = indeterminate)

Bits 7–0 **ADATA7 – ADATA0.** Data Value for pins A7 – A0.
If the pin has been selected as an **input** (*x*DIR*n* = 0), then the value read from the corresponding bit is the value seen on the pin.

0 = Value of selected bit is a 0 (V_{IL}).
1 = Value of selected bit is a 1 (V_{IH}).

If the pin has been selected as an **output,** then the value written to the bit is the value output on the corresponding pin.

0 = Value of selected bit is a 0 (V_{OL}).
1 = Value of selected bit is a 1 (V_{OH}).

3.7 Interrupt and Exception Handling

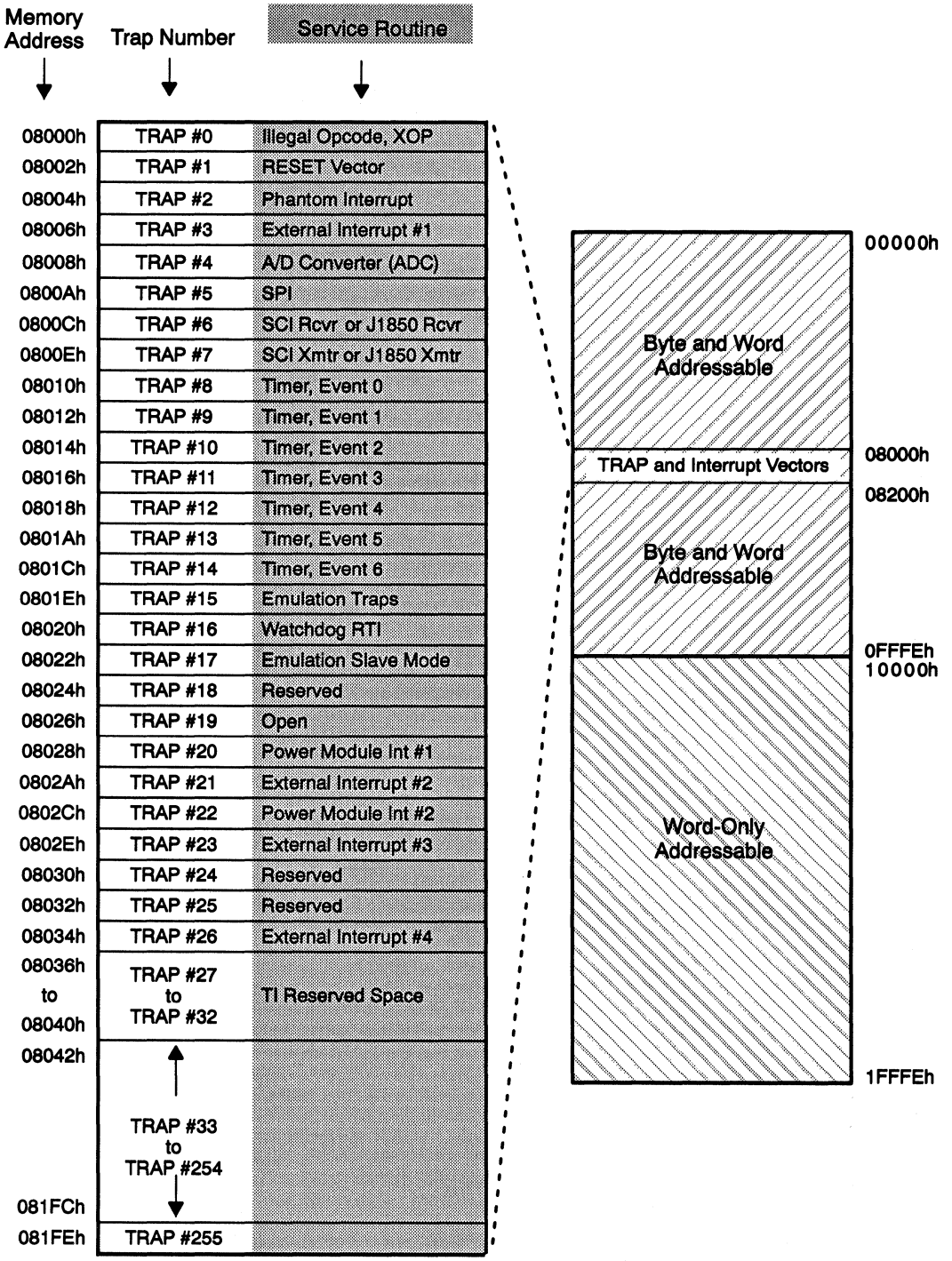
TMS370C16 recognizes four interrupt/exception sources, summarized below. The actual number of interrupt sources, as well as their associated interrupt vector(s), is device specific. This reference guide provides general information for the entire product range of 'C16-based devices. Refer to the specific device data sheet and module reference guide for more information.

3.7.1 Interrupt/Exception Sources

- **Resets** (hardware initiated) are unarbitrated by the CPU and take immediate priority over any other executing functions. All interrupts and the NMI (discussed below) are disabled until being enabled by the reset's service routine (at 08002h in the vector table). Resets are described in further detail in Section 3.2 on page 3-3.
- **Nonmaskable interrupts (NMIs)** (discussed in subsection 3.7.4 on page 3-23) are generated at an external interrupt pin. An NMI takes priority over peripheral interrupts and software exceptions. It can be locked out by an already executing NMI or a reset. Its service routine start address is located in the vector table at 08006h. See the specific device data sheet for more information on devices having more than one NMI.
- **Peripheral interrupts** (discussed in subsection 3.7.5 on page 3-24) are initiated by any of the peripheral modules attached to the CPU. They can be masked off by the L2–L0 interrupt level bits of the ST. Figure 3–5 on page 3-20 illustrates the vector configuration.
- **Software exceptions** (discussed in subsection 3.7.6 on page 3-24) are not arbitrated by the CPU. When these are executing, the ST L2–L0 interrupt level bits are set to all ones (111_2) to mask out peripheral interrupts.
  - A TRAP instruction's vector location corresponds to the trap number in its opcode (0–255). Thus, vector locations range from 08000h for trap 0 up to address 081FEh for trap 255.
  - The other software exceptions (unimplemented opcodes and the ILLEGAL instruction) trap to the address at 08000h.

Whenever an enabled interrupt/exception source requests service, the CPU transfers program flow through a vector that points to the starting address (PC value) of an interrupt/exception subroutine. This context switching transfer is implemented as shown in Figure 3–5:

Figure 3–5. Vector Table Organization in Memory



3.7.2 Vector Table

The vector table (shown in Figure 3–5) contains up to 256 entries, each of which is the starting PC address of an interrupt service routine. The table begins at address 08000h.

When an interrupt is acknowledged, the CPU acquires a vector offset value, which is added to 08000h to locate the corresponding service-routine start address. Each interrupt source is responsible for supplying this offset either through hardware (NMIs and peripheral interrupts) or software (resets, traps, and illegal opcodes).

The single vector table contains the service-routine start addresses for all exceptions and interrupts. Thus, resets, NMIs, and peripheral interrupt vectors are shared with software exception vectors.

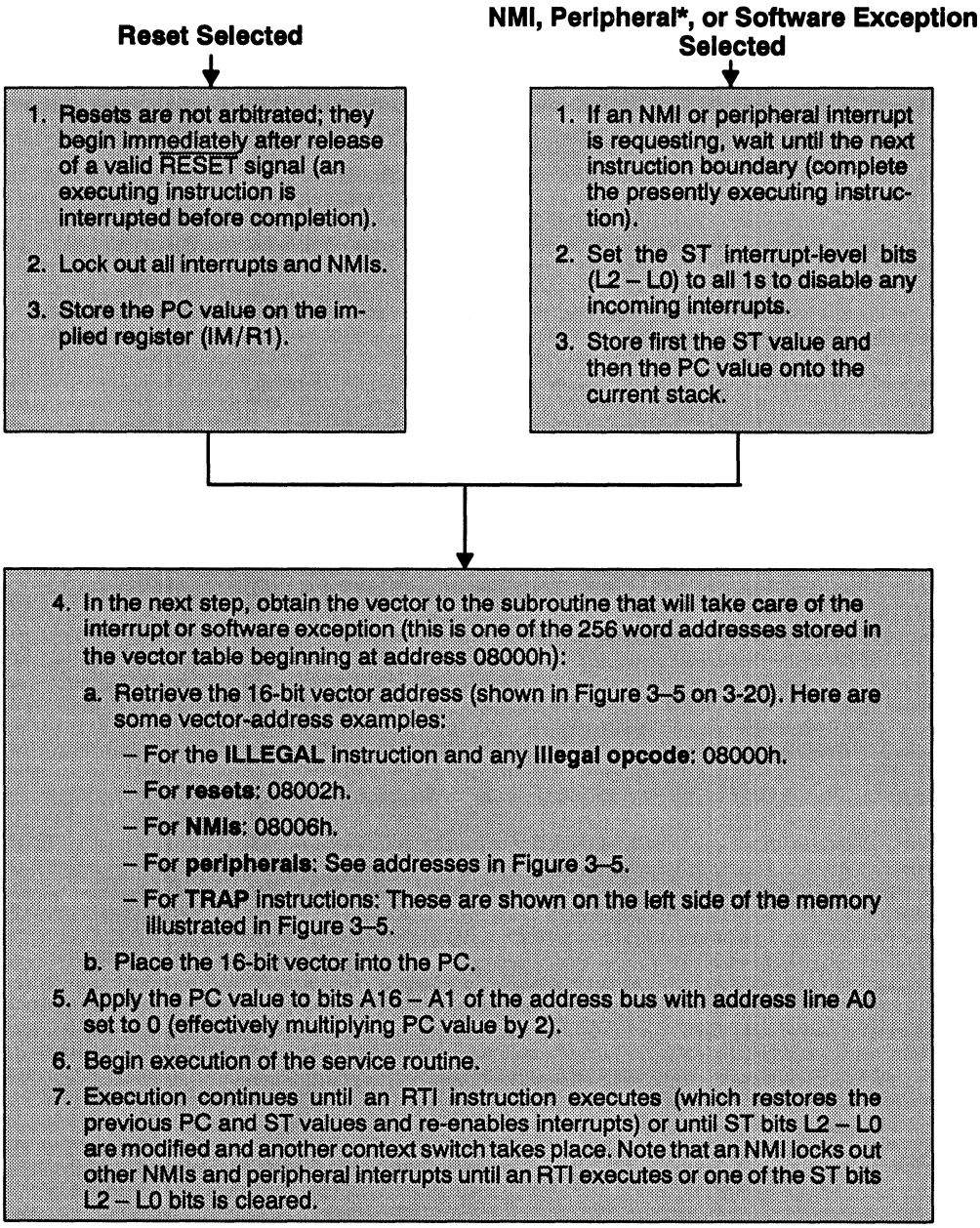
The vector table grows upwards (to higher addresses). The table is only as large as required (but no larger than 512 bytes). The final size of the table is determined by the peripheral module requirements of the device and the application's software use of traps (see your specific device data sheet for size).

The 16-bit address of the first executable instruction in the interrupt handler is a word address that is loaded into the PC and transformed into a 17-bit physical memory address by overlaying bits b15–b0 onto address lines A16–A1 and forcing A0 to 0.

3.7.3 Reset and Interrupt Operation

Figure 3–6 describes the step-by-step sequencing of resets and interrupts.

Figure 3–6. Summary of Reset, NMI, Peripheral Interrupts, and Software Exception Operations



*Peripheral interrupts must be a level *higher* than the level in the ST interrupt-level bits (L2 – L0) in order to execute. Thus, a level of 111_2 locks out any peripheral interrupt.

3.7.4 Nonmaskable Interrupt (NMI) Processing

The NMI is nonmaskable in that it cannot be masked out by the L2 – L0 interrupt-level bits of the status register. However, **NMIs *are* disabled** and will be ignored if:

- An NMI is already executing, **or**
- A reset occurs.

Unless pre-empted by a reset, the NMI will occur on the next instruction boundary if it is internally enabled and a valid external $\overline{\text{NMI}}$ signal is received.

During these two situations, all the ST interrupt-level bits are set to 1, locking out recognition of a pending NMI. Any pending NMI cannot be activated unless one of the following occurs to **(re-)enable NMIs**:

- Execution of an RTI instruction,
- Execution of a TRAP instruction, or
- The clearing to 0 of one or more of the ST register interrupt-level bits (e.g., by an STRI instruction or any other instruction that changes these bits in the ST register—R14).

Also, because ST register interrupt level bits (L2, L1, and L0) are all 1s after an NMI occurs, all interrupt requests are ignored by the CPU until these bits are cleared to zero (changed from their all-1s status).

To summarize, the occurrence of an NMI locks out a pending NMI until the present one is serviced. The RTI instruction is a simple method of re-enabling NMIs, and a pending NMI will be taken following the re-enabling by one of the specified methods.

Avoid Interrupting a Reset With an NMI

It is *imperative* that an NMI not interrupt the CPU during a reset *until* the stack pointer is initialized to a valid value. A valid NMI won't occur as long as an RTI instruction *is not executed* **or** *none of the ST register interrupt-level bits are cleared.*

NMI processing takes several steps:

1) ST → (SP).
2) SP + 2 → SP.
3) PC → (SP).
4) SP + 2 → SP.
5) Look up the vector offset for the NMI trap address (08006h or as specified in the specific device data sheet).
6) Execute NMI interrupt handler at that address.

NMI processing begins with the CPU pushing first its ST register value and then the current PC value onto the stack. The PC points to the word address of the next executable instruction plus two words (four bytes). This is equal to:

$$\frac{\text{17-bit word address bus value +4}}{2} = \textbf{PC value}$$

The PC-value 4-byte offset is due to pipelining prefetch, which leaves the PC pointing four bytes beyond the next opcode, at an instruction boundary.

3.7.5 Peripheral Module Interrupt Processing

Peripheral interrupt requests are maskable by the CPU via the ST register's interrupt-level bits (L2 – L0). During any exception/interrupt processing, these bits are set to 1s, masking off all interrupt requests (except an NMI that was previously enabled; this is explained in subsection 3.7.4).

A request whose level is *greater than* the interrupt-level mask value in the ST register is acknowledged at the next instruction boundary. A request of the *same or lower level* will not be acknowledged.

Execution of an unmasked peripheral interrupt is shown in Figure 3–6, starting on the upper right (page 3-22).

3.7.6 Software Exception (TRAPs, etc.) Processing

A software exception is not arbitrated by the CPU. It occurs when one of the following is executed:

- An illegal opcode
- A TRAP instruction
- An ILLEGAL instruction

During any software exception, the ST register's interrupt-level bits (L2 – L0) are set to 1s, masking off all interrupt requests (except an NMI if NMIs are enabled; this is explained in subsection 3.7.4).

Software exceptions generate their own vector offset value:

- TRAPs use the 8-bit vector offset value (to be added to the vector base address) assembled in the LSbyte of the instruction's opcode.
- The other software exceptions use a vector offset value of 00_{16} — the same as a TRAP #0 instruction.

See descriptions for the TRAP and ILLEGAL instructions in Chapter 5 for further vector information.

3.8 External and Power Module Interrupts

There are three types of external interrupts:

1) External interrupt pins (subsection 3.8.1)
2) Power module fault condition (subsection 3.8.2 on page 3-35)
3) Phantom interrupt controlled exit from an improper interrupt acknowledge sequence (subsection 3.8.3 on page 3-37)

3.8.1 External Interrupt Pins

The 16-byte interrupt frame (shown in Figure 3–7 and Figure 3–8 on the following pages) controls *up* to eight external interrupt pins and *up* to 49 power module interrupts. Pin interrupts can be any of three types: A, B, and C (these are described in Table 3–2 on page 3-26). At least one type A interrupt in INT1 *is required* in any configuration. The actual makeup of the interrupt frame is device specific; *see the device-specific data sheet to determine the interrupt types and control register addresses.*

Rules concerning the 16-byte interrupt frame:

- The first two bytes (addresses 0070h and 0071h) are a type A interrupt (required for *all* interrupt frames).
- The next (higher addressed) 14 bytes can be *any* combination of:
  - Two-byte sets of pin interrupt control/status bits, **and/or**
  - Two-byte sets that contain power-module control/status bits that start at the highest address in the interrupt frame (0007Fh) and are placed contiguously from that address to lower addresses in the frame. Figure 3–7 and Figure 3–8 contain several *examples.*
- The additional Interrupt control/flag bytes are contiguous and follow the type A interrupt bytes that start in addresses 00070h and 00071h, growing to the higher addresses. The first interrupt bytes are INT1 and INT1 FLG bytes, the second are INT2 and INT2 FLG, etc.
- Power module (PM) control and flag bytes start with PM1 at the highest two addresses in the frame (0007Eh and 0007Fh). A second power module (PM2) would be immediately before those for PM1, located at 0007Ch and 0007Dh. PM3 would precede PM2, etc.

Thus, the interrupt frame could contain merely the required single pair of type A interrupt bytes only, as shown in example (a) of Figure 3–7, or a combination of pin interrupts and power module interrupts as shown in examples (b) and (c) in the figure. Example (d) in Figure 3–7 shows pin interrupts in all locations. *The mix and position of interrupt pin types and number of power module pins depends upon device-specific design considerations.*

Figure 3–8 is also a *typical example* of an interrupt frame with all three pin types and their bit names. It also contains two power module control and flag bytes with bit names.

Table 3–2 describes the different external interrupt pin types. All types can be configured for high or low priority, and all interrupt pins are configured to digital inputs on reset. Descriptions of the different types of pins are given in the subsections that follow.

Table 3–2. External Interrupt Types

Pin Type	Configurable as NMI?	Minimum Required	Digital I/O	Freeze Bits[1]	Alternate Functions
Type A	Yes	1	Input only	Yes	V_{PP}/HPO
Type B	Yes	0	I/O	No	—
Type C	No	0	I/O	No	—

[1] Freeze bits are further explained in Section 3.10 on page 3-39.

Table 3–3. External Interrupt Pin Functions

	NMI Bit†	Data Out	Data Dir‡	Polarity§	Priority	Int Enable
nonmaskable Interrupt	1	N/A	N/A	0, 1	N/A	N/A
Interrupt High Priority	0	N/A	N/A	0, 1	0	1
Interrupt Low Priority	0	N/A	N/A	0, 1	1	1
Digital Output '0'	0	0	1	N/A	N/A	0
Digital Output '1'	0	1	1	N/A	N/A	0
Digital Input	0	N/A	0	N/A	N/A	0

† Type C interrupts do not have an NMI bit. Assume a value of 0.
‡ Type A interrupts do not have a data direction bit. Assume a value of 0.
§ Polarity values of 1 and 0 indicate rising and falling edges, respectively.
N/A = Not applicable

Note: INTx Used to Represent INT1 – INT6

In the discussion of interrupt types A, B, and C (subsections 3.8.1.1 through 3.8.1.6 on pages 3-29 to 3-34), the term INTx represents any of the possible interrupt locations (INT1 – INT6) as shown in Figure 3–7 and Figure 3–8. Any of of these interrupt locations can contain any of the three pin-interrupt types (A, B, or C) with one restriction: INT1 in address 00070h must *always* contain a type A.

Figure 3–7. Interrupt-Frame Typical Configurations

70h	INT1	Type A Pin Interrupt
71h	INT1 FLG	Type A Pin Interrupt Flags
72h		Reserved
73h		Reserved
74h		Reserved
75h		Reserved
76h		Reserved
77h		Reserved
78h		Reserved
79h		Reserved
7Ah		Reserved
7Bh		Reserved
7Ch		Reserved
7Dh		Reserved
7Eh		Reserved
7Fh		Reserved

(a) Single Interrupt (Minimum Configuration)

70h	INT1	Type A Pin Interrupt
71h	INT1 FLG	Type A Pin Interrupt Flags
72h	INT2	Type C Pin Interrupt
73h	INT2 FLG	Type C Pin Interrupt Flags
74h	INT3	Type C Pin Interrupt
75h	INT3 FLG	Type C Pin Interrupt Flags
76h		Reserved
77h		Reserved
78h		Reserved
79h		Reserved
7Ah		Reserved
7Bh		Reserved
7Ch	PM2	Power Module 2 Enable
7Dh	PM2 FLAGS	Power Module 2 Flags
7Eh	PM1	Power Module 1 Enable
7Fh	PM1 FLAGS	Power Module 1 Flags

(b) Interrupts and Power Modules

70h	INT1	Type A Pin Interrupt
71h	INT1 FLG	Type A Pin Interrupt Flags
72h	INT2	Type A Pin Interrupt
73h	INT2 FLG	Type A Pin Interrupt Flags
74h		Reserved
75h		Reserved
76h		Reserved
77h		Reserved
78h	PM4	Power Module 4 Enable
79h	PM4 FLG	Power Module 4 Flags
7Ah	PM3	Power Module 3 Enable
7Bh	PM3 FLAGS	Power Module 3 Flags
7Ch	PM2	Power Module 2 Enable
7Dh	PM2 FLAGS	Power Module 2 Flags
7Eh	PM1	Power Module 1 Enable
7Fh	PM1 FLAGS	Power Module 1 Flags

(c) Interrupts and Power Modules

70h	INT1	Type A Pin Interrupt
71h	INT1 FLG	Type A Pin Interrupt Flags
72h	INT2	Type C Pin Interrupt
73h	INT2 FLG	Type C Pin Interrupt Flags
74h	INT3	Type A Pin Interrupt
75h	INT3 FLG	Type A Pin Interrupt Flags
76h	INT4	Type B Pin Interrupt
77h	INT4 FLG	Type B Pin Interrupt Flags
78h	INT5	Type C Pin Interrupt
79h	INT5 FLG	Type C Pin Interrupt Flags
7Ah	INT6	Type B Pin Interrupt
7Bh	INT6 FLG	Type B Pin Interrupt Flags
7Ch	INT7	Type C Pin Interrupt
7Dh	INT7 FLG	Type C Pin Interrupt Flags
7Eh	INT8	Type C Pin Interrupt
7Fh	INT8 FLG	Type C Pin Interrupt Flags

(d) All Interrupts, Mix of All Three Types

Figure 3–8. Typical Interrupt Frame

Addr	Reg Mnem	7	6	5	4	3	2	1	0	Register Shown
00070h	INT1	Reserved	INT1 PIN DATA	INT1 NMI	Reserved		INT1 POLARITY	INT1 PRIORITY	INT1 ENABLE	Type A Interrupt
00071h	INT1 FLG	INT1 FLAG	Reserved							Type A Interrupt Flag
00072h	INT2	Reserved	INT2 PIN DATA	INT2 NMI	INT2 DATA DIR	INT2 DATA OUT	INT2 POLARITY	INT2 PRIORITY	INT2 ENABLE	Type B Interrupt
00073h	INT2 FLG	INT2 FLAG	Reserved							Type B Interrupt Flag
00074h	INT3	Reserved	INT3 PIN DATA	Reserved	INT3 DATA DIR	INT3 DATA OUT	INT3 POLARITY	INT3 PRIORITY	INT3 ENABLE	Type C Interrupt
00075h	INT3 FLG	INT3 FLAG	Reserved							Type C Interrupt Flag
00076h		Reserved								
00077h		Reserved								
00078h		Reserved								
00079h		Reserved								
0007Ah		Reserved								
0007Bh		Reserved								
0007Ch	PM2 ENABLE	PM INT ENA 2	PM STS ENAB 13	PM STS ENAB 12	PM STS ENAB 11	PM STS ENAB 10	PM STS ENAB 9	PM STS ENAB 8	PM STS ENAB 7	Power Module 2 Enable
0007Dh	PM2 FLAGS	PM2 INT FLAG 2	PM INT FLAG 13	PM INT FLAG 12	PM INT FLAG 11	PM INT FLAG 10	PM INT FLAG 9	PM INT FLAG 8	PM INT FLAG 7	Power Module 2 Flags
0007Eh	PM1 ENABLE	PM INT ENA 1	PM STS ENAB 6	PM STS ENAB 5	PM STS ENAB 4	PM STS ENAB 3	PM STS ENAB 2	PM STS ENAB 1	PM STS ENAB 0	Power Module 1 Enable
0007Fh	PM1 FLAGS	PM INT FLAG 1	PM INT FLAG 6	PM INT FLAG 5	PM INT FLAG 4	PM INT FLAG 3	PM INT FLAG 2	PM INT FLAG 1	PM INT FLAG 0	Power Module 1 Flags

3.8.1.1 Type A Interrupt Pins

Type A interrupt pins can be used as nonmaskable interrupts, normal interrupts, or digital input pins. At least one type A interrupt pin is required on each device and must be located in address 00070h (the first byte — INT1 — in the interrupt frame). A corresponding Type A flag bit is contained in the second byte (described in subsection 3.8.1.2). Additional type A interrupts can be implemented on a device's interrupt frame, their location specified by device design (see applicable device data sheet). The example below shows the type A interrupt at INT1. Bits take the name of the interrupt level (INT2, INT3, etc.).

Type A Interrupt (shown in INT1 location)

Bit #	7	6	5	4	3	2	1	0
00070h	Reserved	INT1 PIN DATA	INT1 NMI	Reserved		INT1 POLARITY	INT1 PRIORITY	INT1 ENABLE
		R–0	RWF–0			RWF–0	RW–0	RW–0

R = Read, *W* = Write, *C* = Clear only, *F* = Freeze bit, *–n* = Value after reset (0, 1, x = indeterminate)

Bit 7 **Reserved.**
Writing to this bit has no effect, and a read is undefined.

Bit 6 **INT1 PIN DATA.** Interrupt Pin Data.
This bit reflects the current level on the interrupt pin, regardless of how the interrupt pin is configured.

0 = The pin is a low input.
1 = The pin is a high input.

Bit 5 **INT1 NMI.** Nonmaskable Interrupt Enable.
This bit determines whether or not this pin can generate a nonmaskable interrupt. A freeze bit can be configured to a 1 or 0 on ROM devices at the time of device fabrication (see Section 3.10 on page 3-39).

0 = The pin is a regular interrupt or a digital input.
1 = The pin is a nonmaskable interrupt.

Bits 4 & 3 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

Bit 2 **INT1 POLARITY.** Interrupt Polarity.
This bit determines whether interrupts are generated on the rising or falling edge. A freeze bit can be configured to a 1 or 0 on ROM devices at the time of device fabrication (see Section 3.10 on page 3-39).

0 = The interrupt is generated on a falling edge (high-to-low transition).
1 = The interrupt is generated on a rising edge (low-to-high transition).

Bit 1 **INT1 PRIORITY.** Interrupt Priority.
This bit determines which level interrupt is requested. The bit is ignored if the NMI bit is set.

0 = High-level interrupt. See the specific device data sheet.
1 = Low-level interrupt. See the specific device data sheet.

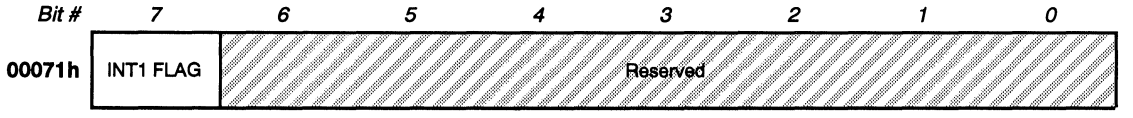
Bit 0 **INT1 ENABLE.** Interrupt Enable.
This bit enables or disables the maskable interrupt. The bit is ignored if the NMI bit is set.

0 = Disable interrupt (use pin as a digital input).
1 = Enable interrupt.

3.8.1.2 Type A External Interrupt Flag Bit

The Type A external interrupt flag bit is the MSB of the byte that accompanies and follows the Type A interrupt pin byte (described in subsection 3.8.1.1). The example below shows INT1 FLAG bit.

Type A External Interrupt Flag (INT1 location)



R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (0, 1, x = indeterminate)

Bit 7 **INT1 FLAG.** Interrupt Flag.
This bit indicates that the selected transition has been detected. It is set, whether the interrupt is enabled or not. The bit can be used for software polling to see whether the selected edge has occurred. It can be cleared by software or a system reset. If used as an interrupt, the bit does not have to be cleared. The interrupt occurs once for each selected edge on the interrupt pin, even though the bit is already set. However, clearing the bit will clear a pending interrupt request from this interrupt pin. The interrupt flag bit is located in a separate register from the interrupt control bits to prevent inadvertent clearing of the flag bit when the control bits are changed with read/modify, write-type instructions such as SBIT0 and SBIT1 (set bit to 0, set bit to 1 instructions).

0 = No transition is detected.
1 = A transition is detected.

Bits 6–0 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

3.8.1.3 Type B Interrupt Pins

Type B interrupt pins can be used as nonmaskable interrupts, normal interrupts, digital input, or digital output pins. Any combination of Type B (as well as Types A or C) interrupt-pin bytes can follow the two Type A interrupt bytes in addresses 00070h and 00071h, as specified by device design (see applicable device data sheet). This Type B interrupt pin byte is followed by a second byte containing the Type B interrupt flag bit (shown in subsection 3.8.1.4).

Type B Interrupt Pin Byte

Bit #	7	6	5	4	3	2	1	0
0007xh	Reserved	INTx PIN	INTx NMI	INTx DATA DIR	INTx DATA OUT	INTx POLARITY	INTx PRIORITY	INTx ENABLE
		R–0	*RW–0*	*RW–0*	*RW–0*	*RW–0*	*RW–0*	*RW–0*

R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (0, 1, x = indeterminate)

Bit 7 **Reserved.**
Writing to this bit has no effect, and a read is undefined.

Bit 6 **INTx PIN.** Interrupt x Pin Data.
This bit reflects the current level on the interrupt pin, regardless of how the interrupt pin is configured.

0 = The pin is a low input.
1 = The pin is a high input.

Bit 5 **INTx NMI.** Nonmaskable Interrupt x Enable.
This bit determines whether or not this pin can generate a nonmaskable interrupt.

0 = The pin is a regular interrupt or a digital I/O.
1 = The pin is a nonmaskable interrupt.

Bit 4 **INTx DATA DIR.** Interrupt x Pin Data Direction.
When this interrupt pin is not enabled as an interrupt, the bit determines whether the pin is a digital input or a digital output.

0 = The pin is an input.
1 = The pin is an output.

Bit 3 **INTx DATA OUT.** Interrupt x Pin Output Data.
When used as a digital output pin, this read/write bit determines whether or not this pin is a 1 or 0.

0 = The pin is a zero if used as a digital output.
1 = The pin is a one if used as a digital output.

Bit 2 **INTx POLARITY.** Interrupt x Polarity.
This bit determines whether interrupts are generated on the rising or falling edge.
0 = The interrupt is generated on a falling edge (high-to-low transition).
1 = The interrupt is generated on a rising edge (low-to-high transition).

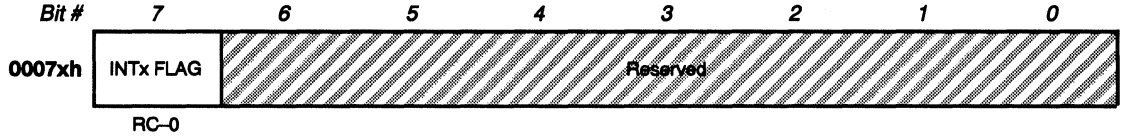
Bit 1 **INTx PRIORITY.** Interrupt x Priority.
This bit determines which level interrupt is requested. The bit is ignored if the NMI bit is set.
0 = High-level interrupt. See the specific device data sheet.
1 = Low-level interrupt. See the specific device data sheet.

Bit 0 **INTx ENABLE.** Interrupt x Enable.
This bit enables or disables the maskable interrupt. This bit is ignored if the NMI bit is set.
0 = Disable interrupt (Use pin as a digital input or output).
1 = Enable interrupt.

3.8.1.4 Type B External Interrupt Flag Bit

This bit is the MSB of the byte following the Type B external interrupt pin byte described in subsection 3.8.1.3.

Type B Interrupt Flag Byte



R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (0, 1, x = indeterminate)

Bit 7 **INTx FLAG.** Interrupt x Flag.
This bit indicates that the selected transition has been detected. It is set, whether or not the interrupt is enabled. This bit can be used for software polling to see whether the selected edge has occurred. It can be cleared only by software or a system reset. If used as an interrupt, the bit does not have to be cleared. The interrupt occurs once for each selected edge on the interrupt pin, even though the bit is already set. Clearing the bit will, however, clear a pending interrupt request from this interrupt pin. The interrupt flag bit is located in a separate register from the interrupt control bits to prevent inadvertent clearing of the flag bit when the control bits are changed with read/modify/write-type instructions such as SBIT0 and SBIT1 (set bit to 0, set bit to 1 instructions).
0 = No transition is detected.
1 = A transition is detected.

Bits 6–0 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

3.8.1.5 Type C Interrupt Pins

Type C interrupt pins can be used as normal interrupts, digital input, or digital output pins. Any combination of Type C (as well as Types A or B) interrupt-pin bytes can follow the two Type A interrupt bytes in addresses 00070h and 00071h, as specified by device design (see applicable device data sheet). This Type C interrupt pin byte is followed by a second byte containing the Type C interrupt flag bit (shown in subsection 3.8.1.6).

Type C Interrupt Pin Byte

Bit #	7	6	5	4	3	2	1	0
0007xh	Reserved	INTx PIN	Reserved	INTx DATA DIR	INTx DATA OUT	INTx POLARITY	INTx PRIORITY	INTx ENABLE
		R–0		RW–0	RW–0	RW–0	RW–0	RW–0

R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (0, 1, x = indeterminate)

Bit 7 **Reserved.**
Writing to this bit has no effect, and a read is undefined.

Bit 6 **INTx PIN.** Interrupt x Pin Data.
This bit reflects the current level on the interrupt pin, regardless of how the interrupt pin is configured.

0 = The pin is a low input.
1 = The pin is a high input.

Bit 5 **Reserved.**
Writing to this bit has no effect, and a read is undefined.

Bit 4 **INTx DATA DIR.** Interrupt x Pin Data Direction.
When this interrupt pin is not enabled as an interrupt, the bit determines whether the pin is a digital input or a digital output.

0 = The pin is an input.
1 = The pin is an output.

Bit 3 **INTx DATA OUT.** Interrupt x Pin Output Data.
When this pin is used as a digital output, this bit determines whether the pin is a 1 or 0.

0 = The pin is a 0 when used as a digital output.
1 = The pin is a 1 when used as a digital output.

Bit 2 **INTx POLARITY.** Interrupt x Polarity.
This bit determines whether interrupts are generated on the rising or falling edge.

0 = The interrupt is generated on a falling edge (high-to-low transition).
1 = The interrupt is generated on a rising edge (low-to-high transition).

Bit 1 **INTx PRIORITY.** Interrupt x Priority.
This bit determines which level interrupt is requested.

0 = High-level interrupt. See the specific device data sheet.
1 = Low-level interrupt. See the specific device data sheet.

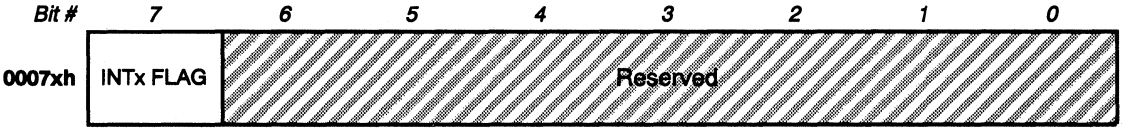
Bit 0 **INTx ENABLE.** Interrupt x Enable.
This bit enables or disables the maskable interrupt.

0 = Disable interrupt (use pin as a digital input or output).
1 = Enable interrupt.

3.8.1.6 Type C Interrupt Flag

This bit is the MSB of the byte following the Type C external-interrupt pin byte described in subsection 3.8.1.5.

Type C Interrupt Flag Byte



R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (0, 1, x = indeterminate)

Bit 7 **INTx FLAG.** Interrupt x Flag.
This bit indicates that the selected transition has been detected. It is set, whether or not the interrupt is enabled. The bit can be used for software polling to see whether the selected edge has occurred. It can be cleared only by software or a system reset. If used as an interrupt, the bit does not have to be cleared. The interrupt will occur once for each selected edge on the interrupt pin, even though this bit is already set. Clearing this bit will, however, clear a pending interrupt request from this interrupt pin. The interrupt flag bit is located in a separate register from the interrupt control bits to prevent inadvertent clearing of the flag bit when the control bits are changed with read/modify/write-type instructions such as SBIT0 and SBIT1 (set bit to 0, set bit to 1 instructions).

0 = No transition is detected.
1 = A transition is detected.

Bits 6–0 **Reserved.**
Writing to these bits has no effect, and reads are undefined.

3.8.2 Power Module Interrupts

Power modules sometimes have fault condition signals that generate interrupts. These signals are routed to the interrupt module. Each interrupt signal has one enable bit and one status flag. Each set of seven internal interrupts has a single interrupt vector. The interrupt level is determined at device fabrication; it cannot be programmed. The power module interrupt registers reside in the same frame as the external interrupt registers.

3.8.2.1 Power Module Interrupt Enable Register

The power module interrupt enable registers contain interrupt enable bits associated with any power modules that are available. See the specific device data sheet to determine availability and naming conventions. Power Module Enable 1 at address 0007Eh is shown as an example. Power Modules 2 and 3, etc., operate identically at their own addresses but are not shown (they follow the numbering scheme shown in Figure 3–8 on page 3-28).

Power Module 1 Enable Interrupt Register (PM1 ENABLE)

Bit #	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0007Eh	PM INT ENA 1	PM STS ENA 6	PM STS ENA 5	PM STS ENA 4	PM STS ENA 3	PM STS ENA 2	PM STS ENA1	PM STS ENA 0
	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	RW–0

R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (*0, 1,* or *x* = indeterminate)

Bit 7 **PM INT ENA 1.** Power Module Interrupt Enable 1.
This bit designates whether or not the seven power module interrupt inputs are able to generate an interrupt request. Note that if this bit is cleared, none of the related seven interrupts in bits 6 – 0 can cause an interrupt. If this bit is set, then an active and enabled interrupt in bits 6 – 0 can cause an interrupt and set the corresponding bit in the power module flag register (described in subsection 3.8.2.2). The PM INT ENA 1 bit provides a quick means to temporarily disable all power module interrupts from a group and then re-enable them using the bit clear (SBIT0) and bit set (SBIT1) instructions. The wakeup signal associated with this interrupt is also disabled when the interrupt is disabled.

0 = Power module interrupt is disabled.
1 = Power module interrupt is enabled.

Bits 6 – 0 **PM STS ENA 6–0.** Power Module Status Interrupt Enable.
These bits specify whether or not the power module interrupt sources are enabled to set the PM INT FLAG 1 bit (subsection 3.8.2.2). To allow an interrupt from a particular power module input, the corresponding PM STS ENA x bit must be set, as well as the PM INT ENA 1 bit.

0 = Power module interrupt is disabled.
1 = Power module interrupt is enabled.

3.8.2.2 Power Module Interrupt Flag Register

The power module interrupt flag registers contain the interrupt status flags associated with the power modules that are used. See the specific device data sheet to determine module availability and naming conventions. Power Module Flag 1 is shown as an example. Power Modules 2 and 3, etc., operate identically at their own addresses but are not shown (they follow the numbering scheme shown in Figure 3–8 on page 3-28).

Power Module 1 Flag Register (PM1 FLAGS)

Bit #	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0007Fh	PM INT FLAG 1	PM INT STS 6	PM INT STS 5	PM INT STS 4	PM INT STS 3	PM INT STS 2	PM INT STS 1	PM INT STS 0
	RC–0	R–x	R–x	R–x	R–x	R–x	R–x	R–x

R = Read, *W* = Write, *C* = Clear only, *–n* = Value after reset (*0, 1,* or *x* = indeterminate)

Bit 7 **PM1 INT FLAG.** Power Module Interrupt Flag.
This bit is set any time one of the power module interrupt status bits is active and its corresponding PM STS ENA x bit is also set. This flag can be cleared only by writing a 0 (writing a 1 has no effect). Values read at this bit:

0 = Power module interrupt has not occurred since flag last cleared.
1 = Power module interrupt has occurred since flag last cleared.

Bits 6 – 0 **PM INT STS 6–0.** Power Module Interrupt Status Flags.
These read-only bits reflect the status of the input source signal to the power module interrupts. If the source is in its active state causing an interrupt, this bit will read a 1; otherwise, it will read a 0.

0 = Power module interrupt is inactive.
1 = Power module interrupt is active.

3.8.3 Phantom Interrupt Vector

The phantom interrupt vector (shown at address 08004h in Figure 3–5 on page 3-20) is a system interrupt integrity feature that allows a controlled exit from an improper interrupt acknowledge sequence. For example, if the CPU receives an interrupt request from a device module, the CPU then reads the priority chain of the device modules to determine which module has a pending interrupt. If the CPU finds no module with a pending interrupt, even though the CPU received an interrupt request, the phantom interrupt vector is accessed. Because this condition is considered to be an invalid operation, it is suggested that the phantom interrupt vector point to a reset generating routine (software reset) so that the device will resume operating from a known condition.

3.9 Multiple Interrupt Servicing

When multiple interrupts are pending simultaneously, the interrupt with the highest level priority is serviced first. This order of service is established through the physical daisy chain connections on the interrupts and by the interrupt mask level in the ST, bits L2–L0.

When servicing an interrupt, the processor automatically sets the interrupt mask bits L2–L0 to 1. This prevents all other interrupts (except an NMI) from being recognized during the execution of the interrupt service routine. If an NMI causes its interrupt service routine to be entered, then even subsequent NMIs are disabled until the NMI interrupt service routine is exited with an RTI (return from interrupt) instruction. Once the service routine is exited, the old status register contents are popped from the stack. This returns the ST interrupt mask bits to their original conditions, thus allowing pending interrupts to be recognized.

An interrupt service routine can allow nested interrupts by modifying the ST interrupt mask bits during interrupt service routine execution. This permits other interrupts to be recognized during the service routine execution. When a nested interrupt service routine completes, it returns to the previous interrupt service routine when the RTI instruction executes. Too many nested interrupts could overflow the stack, causing program failure.

3.10 TMS370C16 Interrupt Configurability Options

The Type A interrupt (described in subsection 3.8.1.1 on page 3-29) allows a *freeze* option regarding:

- ❑ Nonmaskable interrupt functionality
- ❑ Active edge polarity of the interrupt

You can configure your device with freezable control bit mask options during the final stages of the manufacturing process. This freeze option allows you to configure the function of any available Type A interrupt on the device to meet your system requirements. Freezable control bits can be frozen in either a 1 or 0 value. If a control bit is frozen, software control over that bit is disabled, and the Type A interrupt will always operate relative to the frozen state of the bit.

To configure your device with freezable control bit mask options at the time of manufacture, complete a New Code Release Form (NCRF) indicating the desired options. The NCRF is available through any local TI field sales office.

The two control bits in the Type A interrupt control register that can be individually frozen during the manufacturing process are the INT1 NMI (INTx.5) bit and the INT1 POLARITY (INTx.2) bit. Table 3–4 illustrates the possible freeze options available and how Type A interrupt operation is affected.

Table 3–4. Type A Interrupt Control Bit Freeze Options

INT1 NMI (INTx.5)	INT1 POLARITY (INTx.2)	Type A Interrupt Functionality
Writeable	Writeable	Fully software selectable.
0 (Frozen)	Writeable	Type A interrupt can never be configured as an NMI. Polarity is software selectable.
1 (Frozen)	Writeable	Type A interrupt will always be configured as an NMI. Polarity is software selectable.
Writeable	0 (Frozen)	Type A interrupt NMI functionality is software selectable. Polarity is always on the falling edge only.
Writeable	1 (Frozen)	Type A interrupt NMI functionality is software selectable. Polarity is always on the rising edge only.

3.11 Low-Power and Idle Modes

3.11.1 Overview

Low-power modes reduce the operating power by reducing or stopping the internal clock signals used by various modules in the device. There are two types of low-power modes: the *halt* and *standby* modes (see the *Clock Modules Reference Guide* for implementation information.) A third mode, *idle*, is not actually a low-power mode, but a wait state.

The TMS370C16 low-power (powerdown) modes are defined as follows:

- **Halt** mode provides the lowest level of power reduction by stopping all system clocks.
- **Standby** mode provides an intermediate level of power reduction by stopping the system clocks to the CPU. The oscillator and watchdog (if available) clocks are still active in the standby mode.
- **Idle** mode provides no power reduction at all. The CPU in effect, goes into an infinite loop and executes the IDLE instruction until a reset occurs or an enabled interrupt causes another operation to occur.

These modes can be permanently enabled or disabled through mask options for ROM-based devices. If the device has the low-power mode disabled through this mask option, writing to the low-power selection control bits in the oscillator module has no effect. Once the low-power selection control bits are initialized, executing an IDLE instruction causes the device to enter one of the two low-power modes or the *idle* mode.

Note: Low-Power Modes Depend on Oscillator Module

The low-power modes for 'C16 CPU-based devices and the methods of selection depend a great deal on the oscillator module used on the device. See the specific device data sheet and the oscillator module user's guide for more information on the availability and implementation of low-power modes.

3.11.2 Low-Power Wakeup Interrupt

The TMS370C16 CPU-based architecture enables the device to be pulled out of low-power modes through a maximum of 24 selectable actions, as well as any power module interrupt that is present on the device. The actual number and selection of the 24 wakeup actions is device specific. Typically, reset or any enabled external interrupt, as well as any other enabled module interrupt

(SCI, RXD, RTI, etc.), pulls the device out of a low-power mode. See the specific device data sheet to determine exactly which actions allow the low-power modes to be exited.

Remember that even though an interrupt is designed to allow an exit from the low-power mode, that particular interrupt still must be enabled locally and globally to actually bring the device out of the low-power mode. For example, a device can have an SCI available and the SCI RXD interrupt selected to allow low-power mode exit. If the SCI RXD interrupt is disabled locally or if global interrupts are disabled, the low-power mode will not be exited. You must ensure a low-power mode exit path is available before entering a low-power mode.

Chapter 4

Addressing Modes

This chapter describes the addressing modes supported by the TMS370C16 microcontroller instruction set and covers the following topics:

Topic **Page**

4.1 Mode Summary

The various addressing modes of the TMS370C16 CPU and their syntax are described in the pages listed in Table 4–1 below. To find which modes apply to a specific instruction, consult the instruction-set summary table in Section 5.2, beginning on page 5-4.

Table 4–1. Addressing Mode Summary

Addressing Mode	Description	Section	Page
Implied	Operand is not required. Instruction operation is implied in the mnemonic.	4.2	4-3
PC Relative	Operation is relative to the PC contents.	4.3	4-4
Memory Direct	Operation is on a specified memory address.	4.4	4-5
Immediate	Operate on a value specified in the operand.	4.5	4-7
Register Direct	Operate on the value in a register.	4.6	4-8
Register Indirect †	Operate on a value at an address in a register.	4.7	4-9
No Displacement	Register contents = effective address (includes both predecrement and postincrement modes)	4.7.1	4-10
With Displacement	Offset + register contents = effective address (includes extra indirection with CALL and JMP instructions)	4.7.2	4-13

†Section 4.8 (page 4-16) describes how to set the word address in a register for using indirect addressing with the CALL, JMP, and FMOV instructions.

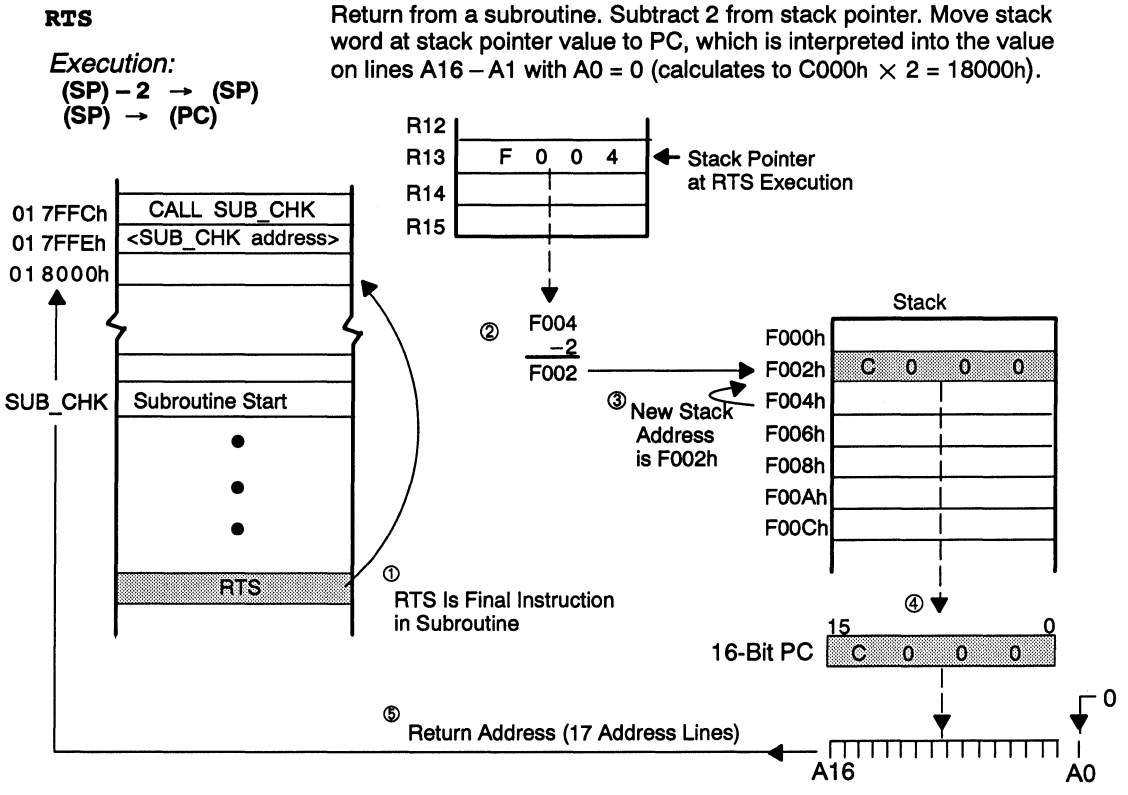
To designate contents, the following apply:

Symbol	Meaning	Example
(Rx) or (x)	Contents of register x or of memory at address x	(R4) or (LABEL)
((x))	Contents of memory designated by contents of x	(disp + (Rn))

4.2 Implied Addressing

This class of instructions does not require you to specify an operand. The operands to be used are predetermined. For example, the implied instruction RTS has two implied operands: the stack pointer (SP) and the program counter (PC). Other instructions using this form of address are RTI (return from interrupt) and UNLINK (unlink and deallocate stack frame).

Figure 4–1. Implied Addressing



Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

4.3 PC-Relative Addressing

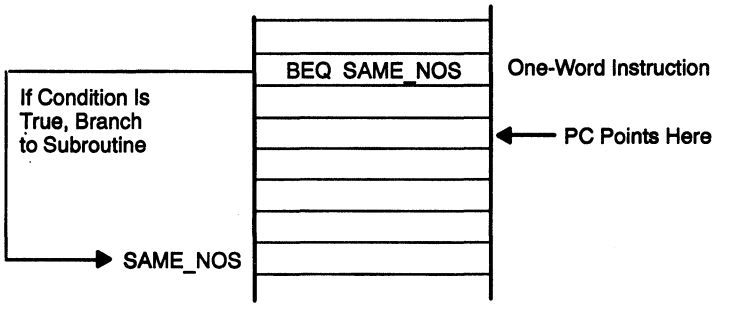
This format adds or subtracts a value from the PC to derive the effective address of the next instruction. Instructions using this format are Bcond, BRBIT0, BRBIT1, and DBNZ.

Figure 4–2. PC-Relative Addressing

BEQ SAME_NOS

Execution:
(PC) + displacement → (PC)
(if condition true)

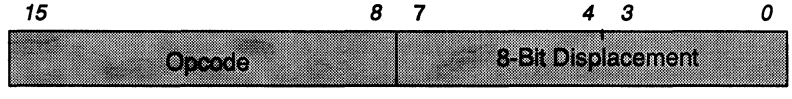
If the status register **Z[[ST]] = 1** (equal condition true), branch to address SAME_NOS by adding 2 × 8-bit displacement to the PC value (presently pointing 2 words beyond the BEQ instruction). This provides a signed displacement of +129 words or –126 words from the BEQ instruction's address. If **Z[[ST]] = 0**, go to the next instruction.



For Bcond, BRBIT0, and BRBIT1, a signed 8-bit value is added to the PC as address lines A8–A1 to redirect execution flow from the executing instruction's 17-bit *physical memory address.* For the DBNZ instruction, a *four-bit unsigned* value in bits 7–4 of the instruction word is *subtracted* from the PC's corresponding value for address lines A4–A1. The following table shows the displacement from the *physical address* of the PC.

Instruction	Maximum Displacement
Bcond (where **cond** represents the condition mnemonic)	+129 words after and –126 words before the physical address of the PC
BRBIT0 and **BRBIT1**	+130 words after and –125 words before the physical address of the PC
DBNZ	Up to –15 words before the PC

The 8-bit displacement is contained in the LSB:



4.4 Memory-Direct Addressing

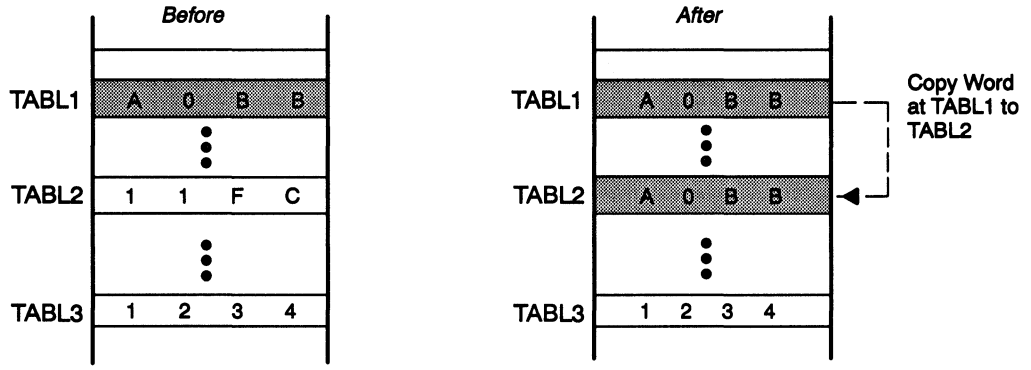
This addressing mode provides an easy way to deal directly with absolute addresses or labeled addresses. It is available *only* for instruction formats in which the indirect register with offset format (*$*disp_{16}$ [Rn]*) is used (as explained in the note on the next page).

Figure 4–3. Memory-Direct Addressing (& Operator)

MOV &TABL1,&TABL2

Execution:
(TABL1) → (TABL2)

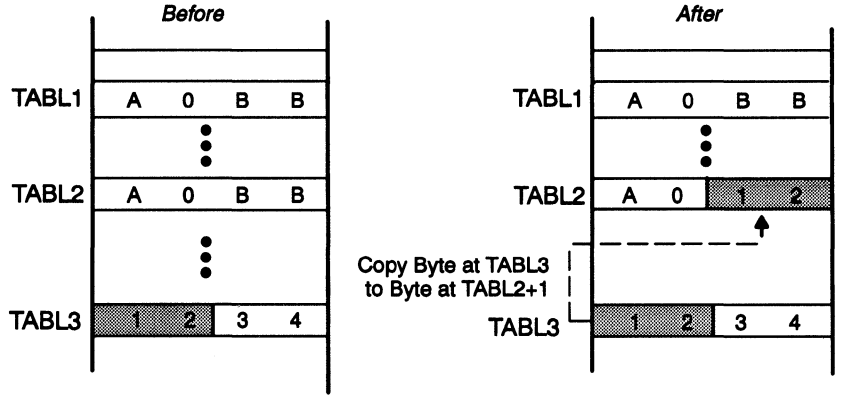
Move (copy) the entire contents (word value) at address TABL1 to address TABL2. Leave the source-address contents unchanged. Consider TABL1 and TABL2 to be on ***even address boundaries*** in order to work correctly with a move-word instruction.



MOVB &TABL3,&TABL2+1

Execution:
$(TABL3_{byte})$ → $(TABL2+1_{byte})$

Move (copy) the byte contents at address TABL3 to the byte at address TABL2+1. Leave the source-address contents unchanged. TABL2 and TABL3 are on ***even address boundaries*** in this example.



Note: Derivation of Memory-Direct Format (& Operator)

The &LABEL-format instruction is derived by transforming the &LABEL operand into the **displacement16[ZR]* format (ZR = R15, the zero register). Thus the zero register value does not change the source or destination address, leaving it equal to the *displacement16* value of LABEL.

For example:

```
MOV     &LABEL,R10
```

is assembled as if written as:

```
MOV     *LABEL[ZR],R10
```

and its timing is the same as for the **disp[Rn]* format.

The second instruction example above moves the contents at LABEL (zero offset) to R10. The corresponding opcode value in this example is 22h, and the instruction needs three cycles to execute, as shown for the formats for the MOV instruction, beginning on page 5-70.

The &LABEL format can be used with *any* instruction that uses the **disp[Rn]* operand (e.g., ADD, ADC, AND, CALL, CLR, etc.).

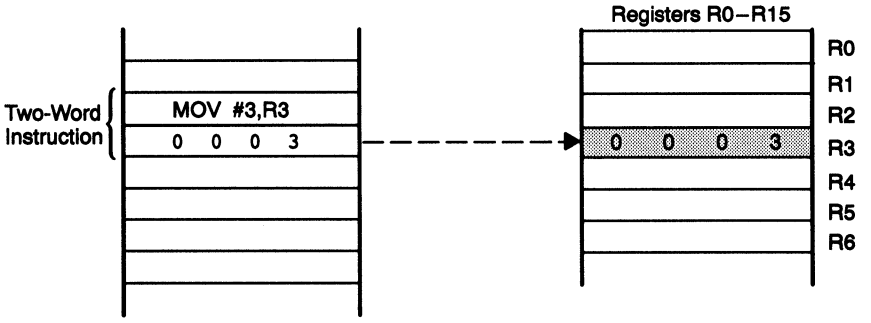
4.5 Immediate Values

This format contains a signed immediate number that will be operated on by the instruction. The immediate value is preceded by an identifying pound sign (#). The different types of immediate instructions are described below.

Figure 4–4. Operand Is Immediate Value (# Operator)

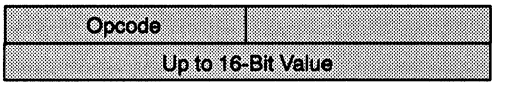
`MOV #3,R3` Move (copy) the immediate value 3 to R3. The immediate value operand is signified by a # prefix.

Execution:
Immediate operand → (R3)



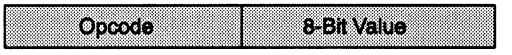
Extension Word (Up to 16 Bits)

A 16-bit extension word following the instruction word contains the immediate value:



Embedded 8-Bit Immediate

The immediate value is in the LSbyte of the instruction word:



Instructions using this format include TBIT0, TBIT1, LINK, RTDU, and TRAP.

Embedded 4-Bit Immediate

The immediate value is in the four MSBs of the instruction word's LSbyte:

Instructions using this format include ADQ, ADQB, MOVQ, SUBQ, SUBQB, STRI, and the shift instructions (SHL, SHLL, ASR, ASRL, ASR0, ASR0L, LSR, and LSRL).

4.6 Register-Direct Addressing

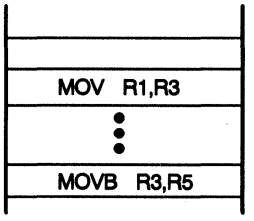
Values within registers are operated upon. The effective address is within the first **64K** bytes *except* for the CALL and JMP instructions, which address **128K** bytes.

Figure 4–5. Register-Direct Addressing

① **MOV R1,R3**
② **MOVB R3,R5**

Move (copy) the entire contents (word value) of R1 to R3. Leave the source register (R1) unchanged. Later, move the LSbyte of R3 to the LSbyte of R5; zero-extend the MSbyte of R5.

Execution:
① **(R1) → (R3)**
② **(R3 LSbyte) → (R5 LSbyte)**
zeroes → (R5 MSbyte)



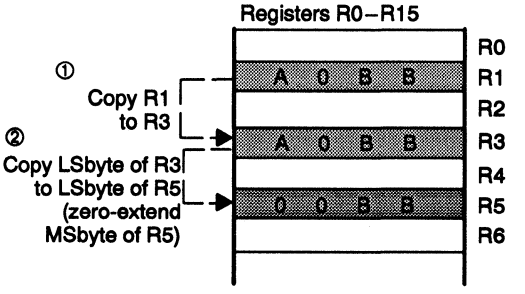


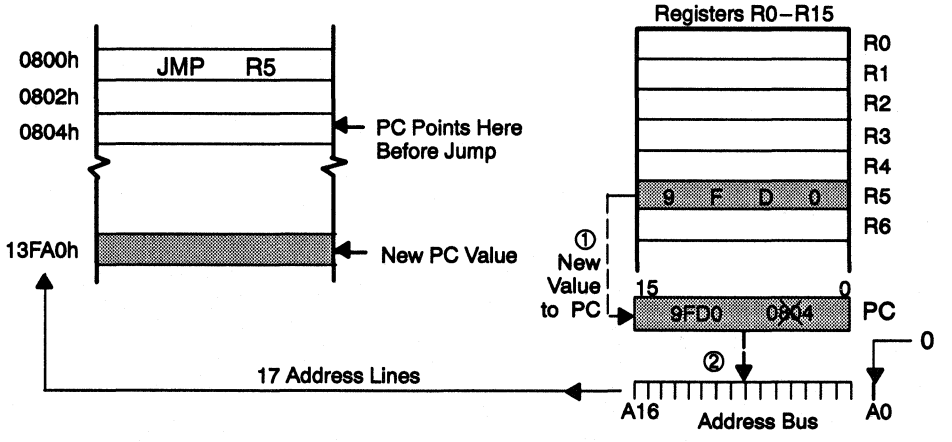
Figure 4–6. Register Direct With CALL or JMP Instructions Addresses 128K Bytes

When used with the CALL or JMP instructions, this mode addresses **128K bytes** (as shown above, address line A16 = 0). For JMP or CALL, place the new 16-bit value into the PC, then overlay the PC value onto address lines A16–A1 with A0 set to 0. Since this essentially multiplies the register contents by two, the register's contents must be half the absolute memory address value. You can use the question mark operator (?) to fill the register with this value (as shown in Section 4.8 on page 4-16).

JMP R5

Jump to the address stored in R5. This address is placed in the PC and then overlaid on the address-bus lines (9FD0h x 2 = 13FA0h). The CALL acts similarly but also provides linkage to the instruction following the CALL.

Execution:
(R5) → (PC)



4.7 Register-Indirect Addressing

The forms of indirect addressing are listed in Table 4–2 below:

Table 4–2. Register-Indirect Addressing Summary

Indirect Addressing Mode	Example Using MOV		Description	See	On Page
No Displacement	MOV	*R1,R2	The effective address of the source is the value in R1. Move (copy) contents at that address to R2.	Figure 4–7	4-10
Predecrement, no displacement	MOV	*–R1,R2	**Before** the move, decrement the contents of R1 by 2 (for word instructions — by 1 for byte instructions). **Then** move (indirect) the contents *at the address in R1* into register R2.	Figure 4–7 Figure 4–8	4-10 4-11
Postincrement, no displacement	MOV	*R1+,R2	**First** move (indirect) the contents *at the address in R1* into register R2. **Then** increment the contents of R1 (by 2 for word instructions — by 1 for byte instructions).	Figure 4–9	4-12
With Displacement	MOV	*DISP[R1],R2	DISP = amount added to R1 to compute the effective address of the source. Move contents at this effective address to R2. Neither predecrement nor postincrement is used with this form.	Section 4.7.2 Figure 4–10 Figure 4–11	4-13 4-13 4-14

Note: *Rn Can Be Used If *disp[Rn] Is Assembled

Several instructions do not provide an indirect register without displacement (**Rn*), but provide an indirect register with displacement (offset) (**disp[Rn]*). However, with such instructions, the assembler accepts **Rn* by assembling the **Rn* format into a **0[Rn]* format.

For example, the assembler statement `ADD *R1,R2`
is assembled as if written `ADD *0[R1],R2`

Thus, the requested instruction becomes a two-word instruction with a zero offset in the second word. In this case, timing is 3 cycles — the cycle count for ADD **disp[Rs],Rd.* (Note that an ADD **Rs,*Rd* operand cannot be used, because there is no ADD **disp[Rs],*disp[Rd]* instruction.)

4.7.1 Register Indirect Addressing, No Displacement (Register Contents = Effective Address)

Register contents point to a memory address that contains the value to be operated on. The register value is treated as a 16-bit memory address (address line A16 = 0) by all instructions except CALL, FMOV, and JMP (which use the value as a *word address* and apply it to the PC, where it is shifted to a 17-bit word address). A method to derive the word address for indirect addressing is shown in Section 4.8 on page 4-16.

Two other forms of indirect addressing are *predecrement* and *postincrement*:

- In postincrement, the register containing the address is first accessed and incremented *afterwards* (see Figure 4–9). This is used with instructions such as MOV, CLR, CMP, STEA, and TST.
- In predecrement, the register containing the address is decremented *before* the address is accessed (see Figure 4–8). This is used with the MOV *–Rs,Rd format.

Note: Decrement/Increment Considerations

1. The value incremented or decremented depends upon the size of the instruction. This value is 2 for word instructions and 1 for byte instructions.
2. When initializing the stack pointer (SP or R14), always write an *even* value to the SP register. An odd value can cause an error.

*Figure 4–7. Register Indirect (Operand: *Rn)*

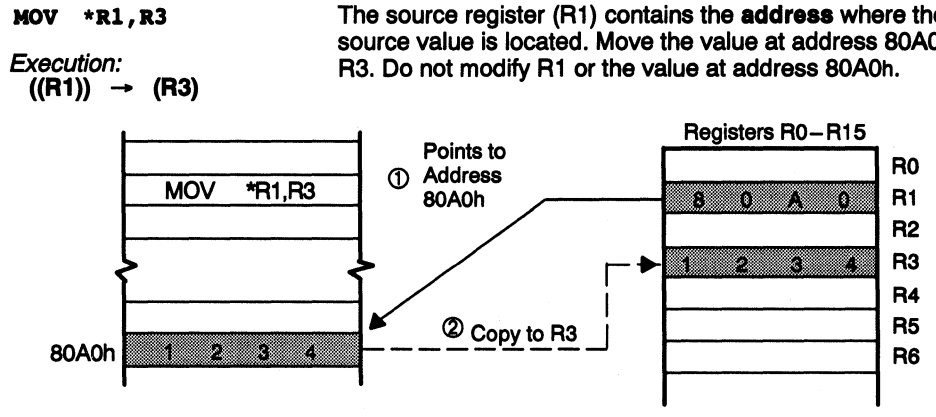
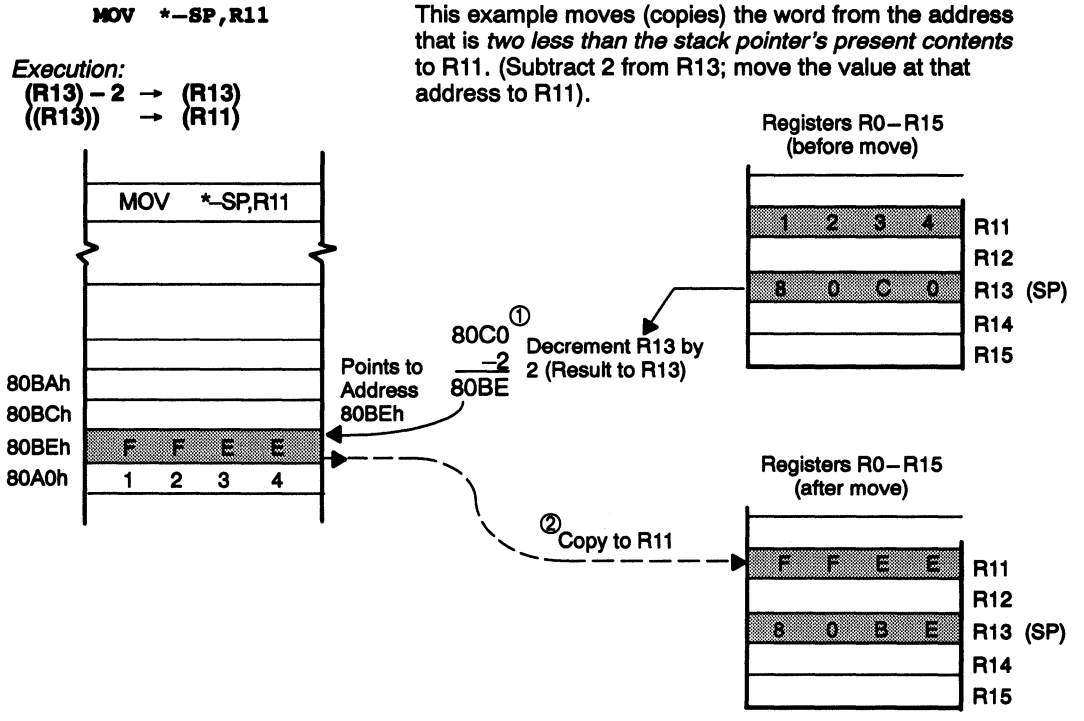


*Figure 4–8. Register Indirect With Predecrement (Operand: *–Rn)*



Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

*Figure 4–9. Register Indirect with Postincrement (Operand: *Rn+) and Predecrement (Operand: *–Rn)*

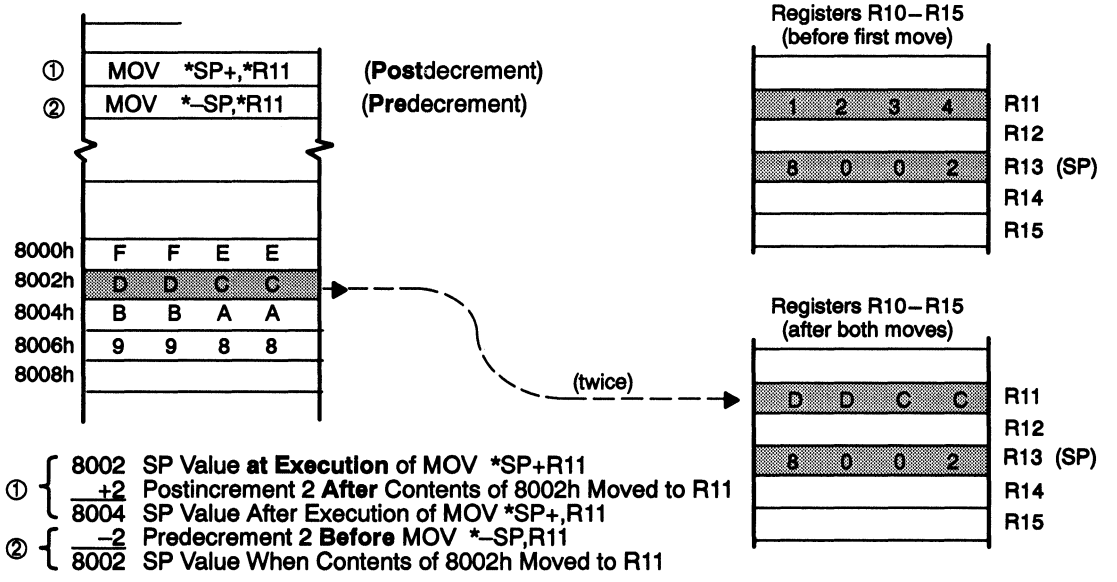
① `MOV *SP+,R11`
② `MOV *–SP,R11`

Execution:

① ((R13)) → (R11)
① (R13)+2 → (R13)
② (R13)–2 → (R13)
② ((R13)) → (R11)

This example demonstrates the execution of both postincrement addressing and predecrement addressing. The two instructions, executed one after the other as shown, repeat exactly the same function: they both move the value at address 8002h to R11.

MOV *SP+,R11 first implements the move, then increments the SP by 2. Then, MOV *–SP,R11 first decrements the SP by two and then repeats the same function. Note that the form of the predecrement instruction shown here (MOV *–Rn,Rn) is the *only* form of the predecrement instruction.



Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

4.7.2 Register Indirect With Displacement (Offset)

These examples show a displacement added to a register's contents to derive the location of the effective address. Figure 4–10 uses **word** format. Figure 4–11 uses **byte** format. Except when used with the JMP or CALL instructions (see Figure 4–12 on page 4-15), indirect addressing is *restricted to the **first 64K** bytes of memory.*

Note that the register to be added to the displacement is contained in **square brackets** (not parentheses).

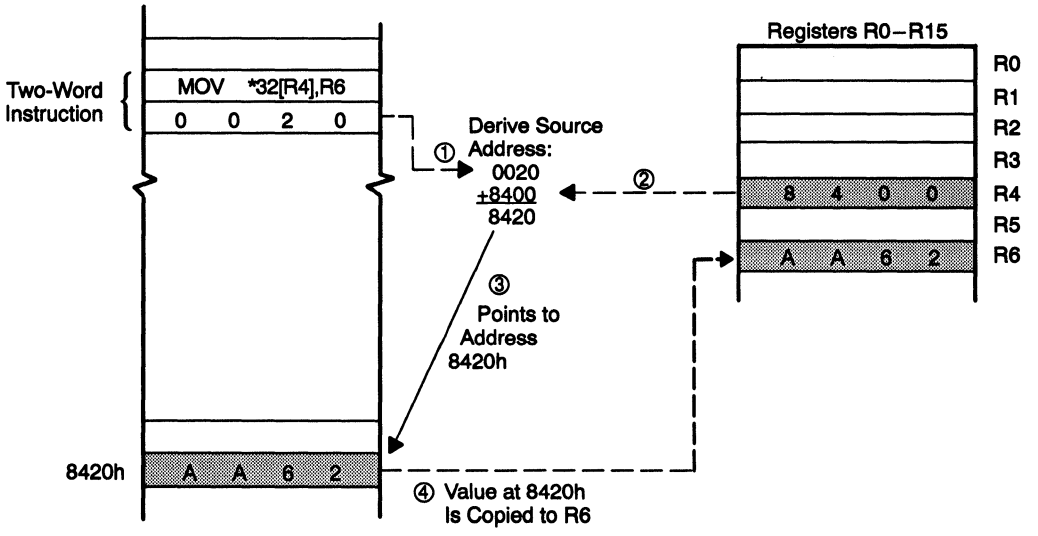
With some instructions (e.g., JMP and CALL), access is to the ***full 128K** bytes of memory.* As shown in Figure 4–13 (page 4-16), these instructions place the value at the resulting effective address into the PC (where it is shifted to create a 17-bit memory address in order to access the full 128K-byte address range).

*Figure 4–10. Offset + Register in Word Format (Operand: *disp16[Rn])*

MOV *32[R4],R6

Execution:
(32 + (R4)) → R6

The source value is found at the address derived by the sum of an immediate displacement value and the contents of the source index register (R4). Thus, move the word value at address 8420h (0020h + 8400h) to R6. Modify only register R6.



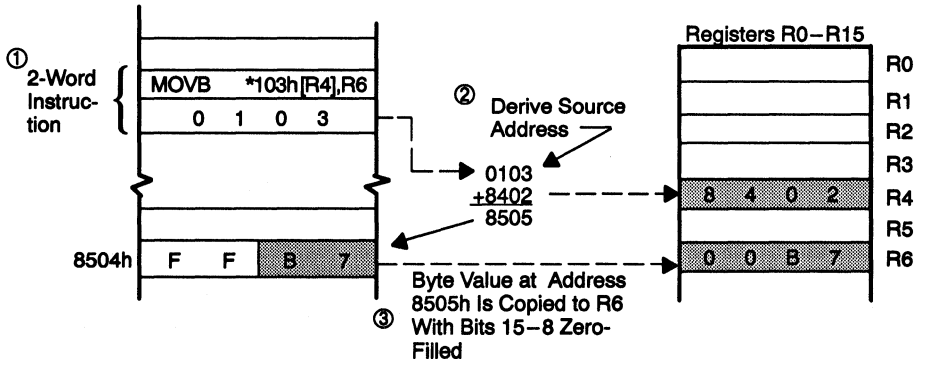
Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

Figure 4–11. Offset + Register in Byte Format (Operand: *disp16[Rn])

MOVB *103h[R4],R6

Execution:
(103h + (R4)) → (R6)

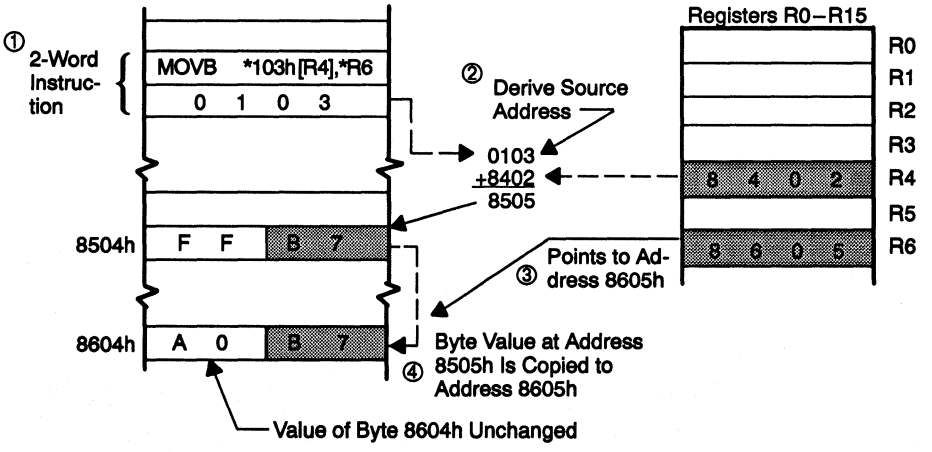
This example is similar to Figure 4–10, except that a byte move is requested (note that the byte is at an uneven address). The source value is found at the address derived by adding the 0103h immediate value and the contents of R4, which contains the 8402h offset. Thus, move the value at address 8505h, which is the LSbyte. However, byte operations extend the byte to a zero-filled word and operate on the word. With a register destination, the entire word is moved to fill the register (a move to a *memory address* changes only the destination byte — see second example below).



MOVB *103h[R4],*R6

Execution:
(103h + (R4)) → ((R6))

The above example is repeated, except that the destination is changed to a memory address because the destination register holds an indirect address. This example shows that the move affects only the designated byte in the destination memory address, leaving any adjacent byte unchanged (*no* zero-filling occurs as it would with a register).



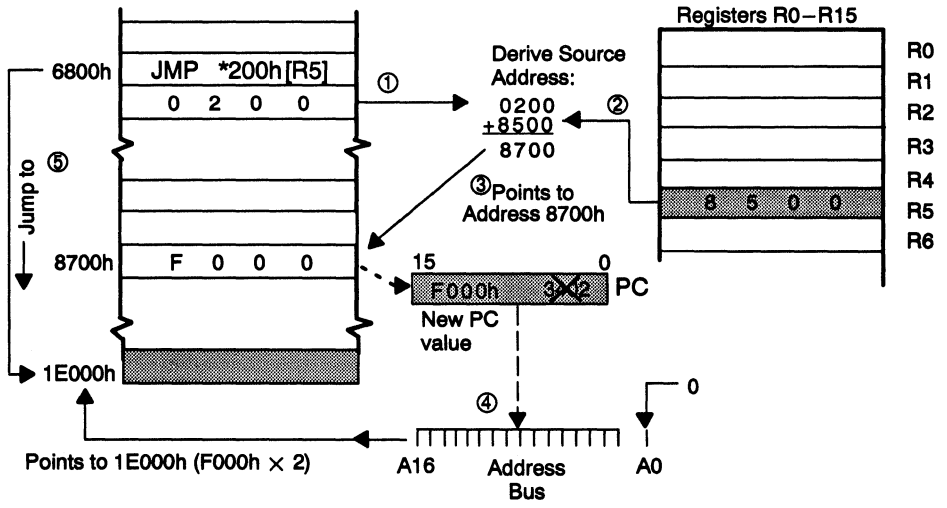
Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

*Figure 4–12. Offset + Register for JMP and CALL Instructions (Operand: *disp16[Rn])*

```
JMP   *200h[R5]
```

Execution:
(200h + (R5)) → (PC))

The destination *word* address (new PC value) is found in a memory address derived by adding the register contents and the offset (displacement) in the operand. This sum (8700h in this example) is a memory address that *contains* the word address (F000h), which is placed in the PC and applied to address lines A16–A1 with A0 held to 0.



Note: A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

The format in Figure 4–12 has an extra level of indirection when used by either the JMP or CALL instruction. The sum of the displacement and register value is a memory address that contains a word address. This word address is placed in the PC and then overlayed on address lines A16–A1 with A0 set to 0 (effectively multiplying the PC value by 2). A method to set the word address for this operation is shown in Section 4.8 on the next page.

Note that with JMP and CALL, indirect register with offset goes to an address to get the final word address. Compare this with the MOV instruction using indirect register with offset for source: the sum of the offset and register is the actual memory address that contains the value to be moved (not the value of another memory address containing the source).

4.8 Setting the Word Address for CALL, JMP, and FMOV Instructions

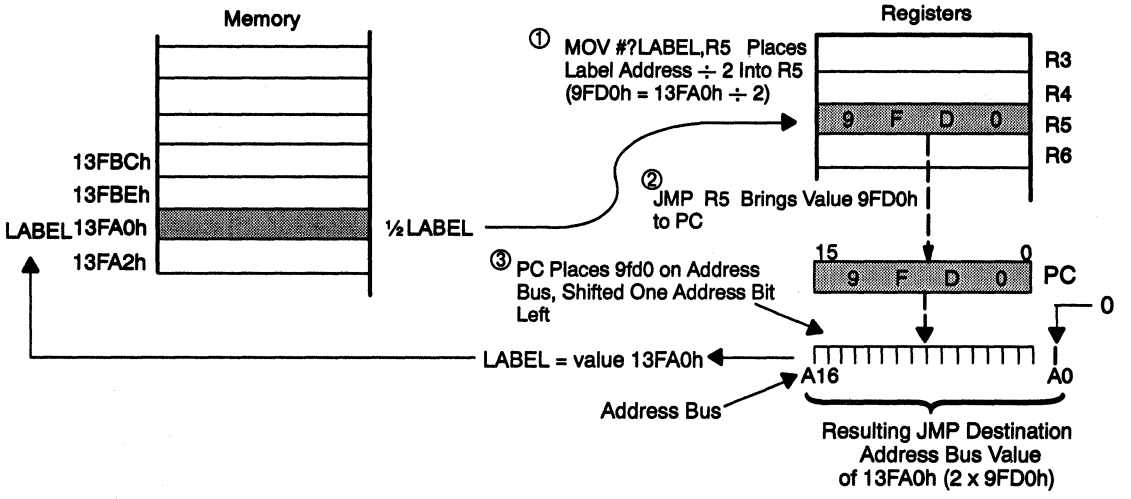
The CALL, JMP, and FMOV instructions address the full 128K-byte address space. They apply their effective operand to address lines A16–A1 with A0 cleared to 0. If you know the 17-bit memory address and want to set up a corresponding word address in a register or memory location, use the question-mark (?) operator, which translates the 17-bit **labeled** memory address into a 16-bit word address (divides the memory address by 2). For example, use the ? operator with a MOV instruction to place the word address into a register. Then use a CALL, JMP, or FMOV instruction to that register or memory location. This is shown in Figure 4–13. This form uses a label representation of the memory address, *not an immediate value.*

Figure 4–13. Using the ? Operator to Set the Word Address for a Direct-Register CALL or JMP

```
MOV    #?LABEL,R5
JMP    R5

Execution:
½ LABEL  →  (R5)
   (R5)  →  (PC)
```

This example shows how the address can be set up for the jump shown in Figure 4–7 on page 4-10. The jump is to a **17-bit memory address**; this means that the value brought into the PC is shifted left one bit (multiplied by 2). Thus, the value brought to the PC must be a 16-bit word address that is **one-half the destination 17-bit memory address.** In this example, the word address is brought in from R5. To get the correct value in R5, use the question-mark (?) operator when loading R5 with the destination address. This operator divides the value by 2, setting up the correct address for the jump destination (as also needed in Figure 4–6). A CALL direct to a register is treated like a JMP.



This method can also be used to set up the indirection register for the FMOV instruction. The bits in the indirection register (either the source or destination) are a word address to be applied to address bits A16—A1.

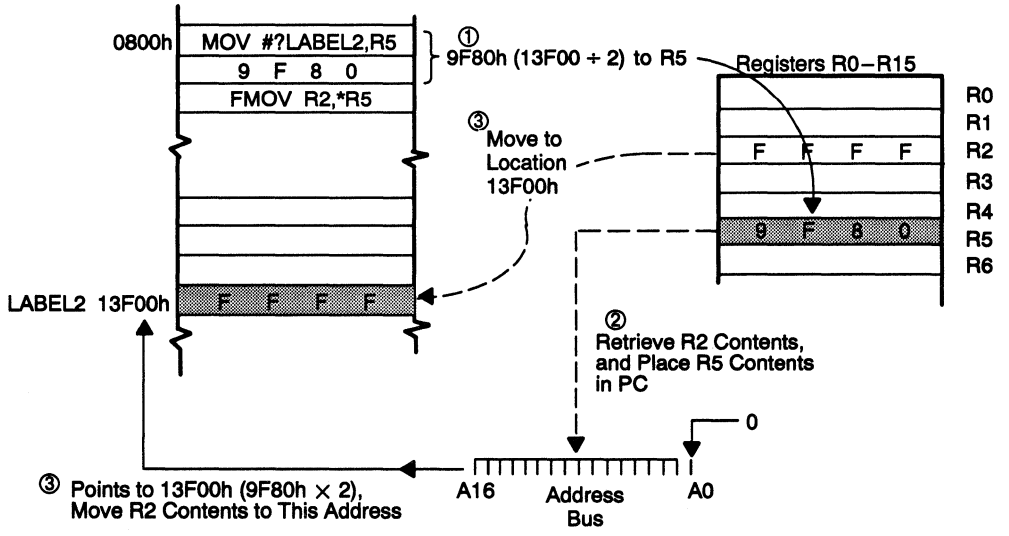
Figure 4–14. Use the ? Operator to Set the Word Address for an Indirect-Register FMOV

```
MOV    #?LABEL2,R5
FMOV   R2,*R5
```

Execution:

½LABEL → (R5)
(R2) → ((R5))

These two instructions set up the word address in a register to be used as the destination for an FMOV instruction. The MOV instruction uses the ? operator to derive the word address for memory address 13F00h (in the second 64K bytes of memory) and have this value ready to be placed in R5 at execution time. The next instruction moves the contents of R2 indirect to this word address in R5.



Chapter 5

Assembly Language Instructions

This chapter describes the mnemonics and operation of the TMS370C16 instruction set, organized in alphabetical order. The chapter begins with a table that summarizes each instruction and auxiliary tables that list the format protocol for the descriptions. Following these are full descriptions of each instruction.

Topic **Page**

5.1 Instruction Set Summary

Section 5.2, starting on page 5-4, summarizes the TMS370C16's instructions. Table 5–1 and Table 5–2 list the abbreviations and symbols used in Section 5.2.

Table 5–1. Abbreviations Used to Describe Instructions

Abbreviation	Meaning
addr16	address; 16 bits in this example
&addr	variant to synthesize direct addressing in memory (assembles as *addr[ZR])
B	byte opcode
C [[ST]]	carry flag in ST
const4, const8	constant (4-bit, 8-bit, etc.)
d, dest	destination
disp8, disp16	displacement (8-bit, 16-bit values shown)
(disp[Rn])	*contents* at the effective address of displacement + value in Rn
enumerator8	member of a list
IEW	instruction extension word
imm	unsigned immediate value; in operand syntax it is preceded by a # symbol; if followd by a number (*imm4*), number = size in bits (compare simm)
IM	implied register (R1)
IW	instruction word
IM:Rd	32-bit concatenation of IM and Rd
FP	frame pointer register (R0)
L	longword opcode
LSB	least significant bit(s)
LSbyte	least significant byte
LSword	least significant word
MSB	most significant bit(s)
MSbyte	most significant byte
MSword	least significant word
N [[ST]]	sign flag in ST
NOT*x*	ones complement of *x*
Op	opcode
OpA	17-bit opcode address (address-bus location)
PC	program counter register
prevA, (prevA)	previous-cycle address bus value; (prevA) = *contents* of previous-cycle address bus value
Rn	register (n = register number, R0–R15)
Rd, Rs, (Rs), (Rd)	registers, destination and source; (Rd) = *contents* of destination register; Rd7 = bit 7 of Rd, etc.
((Rs)), ((Rd))	contents of address contained in Rs or Rd, respectively
R_{FIRST}, R_{LAST}	range of registers

Table 5–1. Abbreviations Used to Describe Instructions (concluded)

Abbreviation	Meaning
*Rn, *Rn+, *–Rn–	indirection, contents of. **Rn* = address value is in *Rn*; **–Rn* = predecrement; **Rn+* = postincrement
Rn(0–7)	bit range within a register (register bits 0–7 in this example)
rtnA	return address
S	S = size of transfer with 1 = byte and 0 = word; see explanation of "b" column for functional logic states in Figure 5–1 on page 5-16.
s, src	source
simm4, simm8	signed immediate value (4-, 8-bits, etc.)
SP	stack pointer register (R13)
ST	status register (R14)
synth. inst	synthetic instruction (synthesized using another assembler format)
vector base address	starting (low) address of the interrupt vectors (an offset is added to this address to determine the address containing the vector of the interrupt)
V[[ST]]	overflow/borrow flag in ST
W	word opcode
ZR	zero register (R15)
Z[[ST]]	zero flag in ST

Table 5–2. Symbols Used to Describe Instructions

Symbol	Meaning
{ }	option to select a value in brackets; for example, {*x, y*} = enter either *x* or *y*, or ADD{B} = ADDB is an optional form of the ADD instruction (add byte vs. add word).
^	bitwise EXCLUSIVE OR (*x^y* = *true* where corresponding bits are different)
~	ones complement (unary): toggle/invert bit values: (0 ↔ 1)
–	negate (twos complement)
<<	left shift (e.g., *(y)* << 6 = shift *y* 6 bit positions to the left)
>>	right shift (e.g., *(x)* >> 4 = shift *x* 4 bit positions to the right)
→	copied to or assigned to
#	immediate operand
( )	contents of. For example, (SP) = contents of stack pointer; (Rd) = contents of Rd.
..	bit selection (*s.bit4* = bit 4 in *s*)
|, +	bitwise OR (*x|y* = *0* if *either x* and *y* = 0)
#?	when a prefix to a label in assembly language, indicates *word address* (one half absolute address)
&	bitwise AND (*x&y* = *0* if *either x* or *y* = 0, but = 1 if *both x* and *y* = 1). If used before a label or address value in assembly language syntax, it indicates direct addressing (synthesizes as *Label[ZR]).
¶	Synthetic instruction

5.2 Instruction Set Summary Table

The following table summarizes each of the TMS370C16's assembly language instructions: mnemonics, operands, opcodes, execution cycles, affect on the status register, and a short description. Included under the Mnemonic column are operands called variants. These are derived by assembling another form of the instruction, usually using a form of the **disp16,[Rn]* operand (explained in the note on page 4-6). Variants can be convenient, but may require more cycles than another format.

Mnemonic	Opcode† B	W	L	Cycles (t_c)	Status‡ Z N C V	Operation Description
ADC					* * * *	Add source plus carry to destination
Rs,Rd		8A		1		*(s) + (d) + (C[[ST]]) → (d)*
**disp16[Rs],Rd*		8B		3		*(an ADD/ADC sequence can be used for 32-bit addition)*
variant:						
&address,Rd		8B		3		
ADD, ADDB					* * * *	Add source to destination
Rs,Rd	31	30		1		*(s) + (d) → (d)*
*Rs,*disp16[Rd]*	33	32		5		
#imm16,Rd	35	34		2		
**disp16[Rs],Rd*	37	36		3		
variants:						
Rs,&address	33	32		5		
&address,Rd	37	36		3		
ADQ, ADQB					* * * *	Add quick immediate to destination
#imm4,Rd	83	82		1		*(s) + (d) → (d)*
#imm4,*disp16[Rd]	85	84		5		*(add short constant — source is 4-bit immediate value in opcode word)*
variant:						
#imm4,&address	85	84		5		
AND, ANDB					* * – 0	Logical AND source with destination
Rs,Rd	41	40		1		*(s) & (d) → (d)*
*Rs,*disp16[Rd]*	43	42		5		
#imm16,Rd	45	44		2		
*#imm16,*disp16[Rd]*	47	46		5		
variants:						
Rs,&address	43	42		5		
#imm16,&address	47	46		5		

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
ASR, ASRL			* * * 0	Arithmetic shift register right
ASR *#imm4,Rd*	B4	*n*+1		(d) >> *n* → (d)
ASRL *#imm4,IM:Rd*	B5	2*n*		
ASR *Rs,Rd*	B6	*n*+3		(arithmetic right shift — source contains
ASRL *Rs,IM:Rd*	B7	2*n*+2		shift count *n*)
ASRL *Rs,IM:Rd* where *Rs*=xxx0h	B7	3		
ASR0, ASR0L			* * * 0	Arithmetic right shift, round to 0:
ASR0 *#imm4,Rd*	B8	*n*+2		(d) >> *n* → (d)
ASR0L *#imm4,IM:Rd*	B9			IF N[[ST]] = 1 and a 1 is shifted out
N[ST] = 0	B9	2*n*+3		of LSB,
N[ST] = 1	B9	2*n*+2		THEN Rd + 1 → Rd.
ASR0 *Rs,Rd*	BA	*n*+4		
ASR0L *Rs,IM:Rd*	BB			*(arithmetic right shift — source contains*
N[ST] = 0	BB	2*n*+3		*shift count n — round to 0)*
N[ST] = 0 (*Rs* = 0)	BB	3		
N[ST] = 1	BB	2*n*+4		
N[ST] = 1 where (*Rs* = xxx0h)	BB	3		
B{COND} *disp8*			– – – –	IF *cond* = *true*, branch to PC + *disp8;*. otherwise, execute next instruction.
BCOND Branch Condition				*Branch conditions; ST bit combinations:*
BC if carry set	C2§			C = 1
BEQ if equal	C3			Z = 1
BGE if greater than or equal	C9	3 if		N^V = 0
BGT if greater than	C7	branch		Z | (N^V) = 0
BHI if higher	C5	taken		C | Z = 0
BHS if higher or same	C1§			C = 0
BLE if less than or equal	C8	2		Z | (N^V) = 1 [Z OR (N XOR V) =1]
BLO if lower	C2§	if		C = 1
BLS if lower or the same	C6	branch		C | Z = 1
BLT if less than	CA	not		N^V = 1
BN if negative	CF	taken		N = 1
BNC if carry clear	C1§			C = 0
BNE if not equal	C4			Z = 0
BNV if overflow clear	CC			V = 0
BP if positive	CD			N | Z= 0
BPZ if plus (not negative)	CE			N = 0
BR always	C0			– –
BV if overflow set	CB			V = 1

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
§ Two pairs of branch instructions have the same opcodes: BHS and BNC are C1h, and BC and BLO are C2h.

Mnemonic		Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
BRBIT0 *#imm3,&addr,disp8*	*imm3* 0 1 2 3 4 5 6 7	 D0 D1 D2 D3 D4 D5 D6 D7	5 (branch taken) 4 (branch not taken)	– – – –	Branch if bit is 0. Test bit *imm3* in byte *addr.* IF bit = 0, branch to PC + *disp8*; THEN (PC) + disp8 → (PC) ELSE, execute next sequential instruction. (The *imm3* value is contained in the 3 LSBs of the opcode.)
BRBIT1 *#imm3,&addr,disp8*	*imm3* 0 1 2 3 4 5 6 7	 D8 D9 DA DB DC DD DE DF	5 (branch taken) 4 (branch not taken)	– – – –	Branch if bit is 1. Test bit *imm3* in byte *addr.* IF bit = 1, branch to PC + *disp8;* THEN (PC) + disp8 → (PC) ELSE, execute next sequential instruction. (The *imm3* value is contained in the 3 LSBs of the opcode.)
CALL *Rd* *addr* **disp16[Rd]* *variants:* **Rd* *&address*		 EB EC ED ED ED	 5 4 5 5 5	– – – –	Jump to subroutine, with linkage CALL *Rd*: Next Instruction Address → (SP) (SP) + 2 → (SP) *(Rd)* → (PC) CALL **Rd* is assembled as CALL **0[Rd].* CALL *&address* is assembled as CALL **address[RZ].* (Both variant forms expect a word address at the destination.)
CLR, CLRB¶ *Rd* **Rd* **Rd+* **disp16[Rd]* *variant:* *&address*		 03 02 05 04 07 06 09 08 09 08	 1 2 2 3 3	1 0 – 0	Clear destination: 0 → *(d)* Synthesized as MOV *ZR,d.*

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
¶ Synthetic instruction

Mnemonic	Opcode† B	W	L	Cycles (t_c)	Status‡ Z N C V	Operation Description
CMP, CMPB					* * * *	Compare source to destination:
Rs,Rd	61	60		1		*(d) – (s)*
#imm16,Rd	63	62		2		and set ST bits accordingly.
**disp16[Rs],Rd*	65	64		3		
**Rs+,Rd*	67	66		3		
**disp16[Rs],*disp16[Rd]*	69	68		5		
variants:						
&address,Rd	65	64		3		
*&address,*disp16[Rd]*	69	68		5		
**disp16,[Rs],&address*	69	68		5		
&address1,&address2	69	68		5		
CMPC					* * * *	Compare source to destination:
Rs,Rd		8E		1		*(d)* – ((*s* – C[[ST]]))
**disp16[Rs],Rd*		8F		3		and set ST bits accordingly.
variant:						
&address,Rd		8F		3		
COMPL, COMPLB¶					* * * *	Twos-complement (negate) destination
Rn	2F	2E		1		(ZR) – *(Rn)* → *(Rn)*
						Synthesized as SUBR *Rn,ZR.*
DBNZ *Rs,disp4*		A8		4 (branch taken) 3 (branch not taken)	– – – –	Decrement register; branch if not 0: *(Rs)* – 1 → *(Rs)* IF *Rs* ≠ 0, branch to PC – *disp4* IF *Rs* = 0, execute next sequential instruction without branching.
DEC, DECB¶					* * * *	Decrement destination
Rd	87	86		1		*(d)* – 1 → *(d).*
**disp16[Rd]*	89	88		5		Synthesized as SUBQ *#1,destination.*
variant:						
&address	89	88		5		
DIVS, DIVSL					* * 0 *	Signed division:
DIVS *Rs,Rd*		A2		2–27▲		*(d)* ÷ *(Rs)* → *(Rd)* (quotient),
DIVSL *Rs,IM:Rd*			A3	2–29		remainder → *(IM).*
DIVU, DIVUL					* * * 0	Unsigned division:
DIVU *Rs,Rd*		A0		3–21♦		*(d)* ÷ *(Rs)* → *(Rd)* (quotient),
DIVUL *Rs,IM:Rd*			A1			remainder → *(IM).*

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
¶ Synthetic instruction
§ Two pairs of branch instructions have the same opcodes: BHS and BNC are C1h, and BC and BLO are C2h.
▲ DIVS takes 2–27 cycles, with two exceptions explained in the instruction's detailed description.
DIVSL takes 2–29 cycles,with eight exceptions explained in the instruction's detailed description.
♦ DIVU and DIVUL take 3–21 cycles, with the exceptions explained in the instruction's detailed description.

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
EXTS, EXTSB EXTS *IM:Rd* EXTSB *Rd*	 AA AB	 2 1	* * * 0	Extend sign of register value: bit 15 value → bits 16 to 31 (word) bit 7 value → bits 8 to 15 (byte)
EXTZ, EXTZB¶ *EXTZ IM:Rd* *EXTZB Rd* 	 02 03	 1 1	* * – 0	Extend (zero fill) register to next larger data size (byte → word *or* word → double word). Synthesized as MOV *ZR,IM* (word) and MOVB *Rd,Rd* (byte).
FMOV *Rs,*Rd* **Rs,Rd*	 F2 F3	 5 5	* * – 0	Move *far*; indirect register accesses 128K bytes: *(Rs)* → *((Rd))* *((Rs))* → *(Rd)*
IDLE	FE	2	– – – –	Idle CPU (reaches idle state in 2 cycles).
ILLEGAL	00	7	– – – –	Generate trap #0 exception. (ST) and (PC) of next instruction → stack; ones → (L2–L0[[ST]])
INC, INCB¶ *Rd* **disp16[Rd]* *variant:* *&address*	 83 82 85 84 85 84	 1 5 5	* * * *	Increment destination *(d)* + 1 → *(d)* Synthesized as ADQ *#1,destination*
INTPU *Rs,IM:Rd* *if IM ≤ Rd* *if IM > Rd*	7D	 9 10	* * 0 0	Perform a rounded straight-line interpolation between values in IM and Rd using interpolation fraction in Rs.
JMP *Rd* *addr* **disp16[Rd]* *variant:* **Rd* *&address*	 E8 E9 EA EA EA	 3 3 4 4 4	– – – –	Jump to destination: *(d)* → (PC). JMP **Rd* is assembled as JMP **0[Rd]*. JMP *&address* is assembled as JMP **address[RZ]*. (Both expect a word address as the destination.)
LDBIT, LDBITB *#imm4,Rd* *#imm4,*disp16[Rd]* *Rs,Rd* *Rs,*disp16[Rd]* *variants:* *#imm4,&address* *Rs,&address*	 94 95 E4 E5 95 E5	 2 4 3 5 4 5	– – * –	Read bit number *s* in *d*: (Bit in *d*) → (C[[ST]]).
LDEA **disp16[Rs],Rd* *variant:* *&address,Rd*	 F0 F0	 2 2	– – – –	Load effective address: *((disp1 + (Rs))* → *(Rd).*

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
¶ Synthetic instruction

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
LIMHS, LIMHSB			* * * *	Limit *Rd* to highest signed *legal* value (in *s*):
**disp16[Rs],Rd*	59 58			IF (V[[ST]]) = 1 *and* (N[[ST]]) = 1 *or*
If V[[ST]] = 1		5		IF (V[[ST]]) = 0 *and (s) < (Rd)*,
If V[[ST]] = 0		6		THEN *(s)* → *(Rd)*, 0 → (V[[ST]]) *and*
variants:				1 → (C[[ST]]).
&address,Rd	59 58			
If V[[ST]] = 1		5		
If V[[ST]] = 0		6		
LIMHU, LIMHUB			* * 0 *	Limit *Rd* to highest *un*signed *legal* value (in *s*):
**disp16[Rs],Rd*	5B 5A			IF (C[[ST]]) = 1 *or*
If C[[ST]] = 1		4		IF *(s) < (Rd)*,
If C[[ST]] = 0		5		THEN *(s)* → *(Rd) and* 1 → (V[[ST]])
variants:				ENDIF
&address,Rd	5B 5A			0 → (C[[ST]])
If C[[ST]] = 1		4		IF an LIMHUB instruction (byte),
If C[[ST]] = 0		5		THEN 0 → *Rd 8– 15.*
LIMLS, LIMLSB			* * * *	Limit *Rd* to lowest signed value:
**disp16[Rs],Rd*	5D 5C			IF (V[[ST]]) = 1 *and* (N[[ST]]) = 0, *or*
If V[[ST]] = 1		5		IF (V[[ST]]) = 0 *and(s) >(Rd)*,
If V[[ST]] = 0		6		THEN *(s)* → *(Rd)*, 0 → (V[[ST]]) *and*
variants:				1 → (C[[ST]]).
&address,Rd	5D 5C			
If V[[ST]] = 1		5		
If V[[ST]] = 0		6		
LIMLU, LIMLUB			* * * *	Limit *Rd* to lowest unsigned value:
**disp16[Rs],Rd*	5F 5E			IF (C[[ST]]) = 1 *or* (source) > (Rd),
If C[[ST]] = 1		5		THEN *(s)* → *(Rd)* and 1 → (V[[ST]])
If C[[ST]] = 0		6		ENDIF
variants:				0 → (C[[ST]]).
&address,Rd	5F 5E			
If C[[ST]] = 1		5		
If C[[ST]] = 0		6		
LINK			– – – –	Link frame pointer to stack pointer:
disp8	F7	4		(FP) → ((SP))
				(SP) → (FP)
				(SP) + 2 → (SP)
				(SP) + 2 x disp8 → (SP)

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)

Mnemonic	Opcode† B	W	L	Cycles (t_c)	Status‡ Z N C V	Operation Description
LSR, LSRL					** *0	Logically right shift *(Rd)* by the count *n* in *s:* *(Rd) >> n → (Rd)*
LSR *#imm4,Rd*		BC		*n*+1		
LSRL *#imm4,IM:Rd*			BD	2*n*		
LSR *Rs,Rd*		BE		*n* + 3		
LSRL *Rs,IM:Rd*			BF	2*n* + 2		
LSRL *Rs,IM:Rd* (where Rs=xxx0h)			BF	3		
MOV, MOVB					* * – 0	Copy the source; place copy in destination: *(s) → (d)*
Rs,Rd	03	02		1		
*Rs,*Rd*	05	04		2		
*Rs,*Rd+*	07	06		2		
*Rs,*disp16[Rd]*	09	08		3		
**Rs,Rd*	0B	0A		2		
**Rs,*Rd*	0D	0C		3		
**Rs,*Rd+*	0F	0E		3		
**Rs,*disp16[Rd]*	11	10		4		
**Rs+,Rd*	13	12		3		
**Rs+,*Rd*	15	14		3		
**Rs+,*Rd+*	17	16		4		
**Rs+,*disp16[Rd]*	19	18		4		
#imm,Rd16	1B	1A		2		
*#imm,*Rd16*	1D	1C		3		
*#imm,*Rd16+*	1F	1E		3		
*#imm,*disp16[Rd]*	21	20		4		
**disp16[Rs],Rd*	23	22		3		
**disp16[Rs],*Rd*	25	24		4		
**disp16[Rs],*Rd+*	27	26		4		
**disp16[Rs],*disp16[Rd]*	29	28		5		
**–Rs,Rd*	2B	2A		3		
variants:						
Rs.&address	09	08		3		
**Rs,&address*	11	10		4		
**Rs+,&address*	19	18		4		
**disp16[Rs],&address*	29	28		5		
#imm,&address						
&address,Rd	21	20		4		
*&address,*Rd*	23	22		3		
*&address,*Rd+*	25	24		4		
*&address,*disp16[Rd]*	27	26		4		
&address1,&address2	29	28		5		
	29	28		5		
MOVQ					* 0 – 0	*imm4 → (Rd)*
#imm$_4$,Rd		80		1		

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
¶ Synthetic instruction

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
MPYBWU *Rs,Rd*	AC	7	* * 0 0	Unsigned 8-bit x 16-bit multiply with rounding: [(*Rs LSbyte*) × (*Rd*) + 80h] ÷ 256 → *(Rd).*
MPYS, MPYSB MPYSB *Rs,Rd* *Rd* ≥ 0 *Rd* < 0 MPYS *Rs,IM:Rd* *Rd* ≥ 0 *Rd* < 0	 A7 A6 	 10 11 13 14	* * 0 0	Multiply signed: *(Rs)* × *(d)* → *(d).*
MPYU, MPYUB MPYUB *Rs,Rd* MPYU *Rs,IM:Rd*	 A5 A4	 8 13	* * 0 0	Multiply *un*signed: *(Rs)* × *(d)* → *(d).*
NOP¶	92	1	– – – –	No operation 0 → (ZR) Synthesized as SBIT0 *#15,ZR*
NOT, NOTB¶ *Rd*	 2D 2C	 1	* * – 0	Ones complement the destination ~*(Rd)* Synthesized as XNOR *ZR,Rd*
OR, ORB *Rs,Rd* *Rs,*disp16[Rd]* *#imm16,Rd* *#imm16,*disp16[Rd]* *variants:* *Rs,&address* *#imm16,&address*	 49 48 4B 4A 4D 4C 4F 4E 4B 4A 4F 4E	 1 5 2 5 5 5	* * – 0	Logical inclusive OR source with dest: *(s)* | *(d)* → *(d).*
POP *R* LAST,*R* FIRST	FA	1 + 2*n* (*n* = repeat cycles)	– – – –	Pop registers from the stack: FOR *index* = Register_Last TO Register_First BY –1, DO (SP) – 2 → (SP) ((SP)) → (register(*index*)).
PUSH *R*FIRST,*R*LAST	F9	1 + *n* (*n* = repeat cycles)	– – – –	Push register values onto the stack: FOR *index* = Register_First TO Register_Last BY +1, DO (register(*index*)) → ((SP)) (SP) + 2 → (SP).

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
¶ Synthetic instruction

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
RTDU *disp8*	F8	5	– – – –	Return from subroutine, unlink stack: (FP) – 2 → (SP) ((FP)) → (FP) ((SP)) → (PC) ((SP) – 2 × *disp8*) → (SP).
RTI	FC	6	* * * *	Return from interrupt: (SP) – 2 → (SP) ((SP)) → (PC) (PC) – 2 → (PC) (SP) – 2 → (SP) ((SP)) → (ST).
RTS	FB	4	– – – –	Return from subroutine: (SP) –2 → (SP) ((SP)) → (PC).
SBB *Rs,Rd* **disp16[Rs],Rd* *variant:* *&address,Rd*	 8C 8D 8D	 1 3 3	* * * *	Destination minus source and carry: *(d)* – *(s)* – (C[[ST]]) → *(d)*. Subtract *s* and carry bit from *d*.
SBIT0, SBIT0B SBIT0 *#imm4,Rd* SBIT0B *#imm4,*disp16[Rd]* SBIT0 *Rs,Rd* SBIT0B *Rs,*disp16[Rd]* *variants:* SBIT0B *#imm4,&address* SBIT0B *Rs,&address*	 92 93 E2 E3 93 E3	 1 5 2 6 5 6	– – – –	Set bit to 0: 0 → bit in *d*. *(Value in s designates bit to clear.)*
SBIT1, SBIT1B SBIT1 *#imm4,Rd* SBIT1B *#imm4,*disp16[Rd]* SBIT1 *Rs,Rd* SBIT1B *Rs,*disp16[Rd]* *variants:* SBIT1B *#imm4,&address* SBIT1B *Rs,&address*	 90 91 E0 E1 91 E1	 1 5 2 6 5 6	– – – –	Set bit to 1: 1 → bit in *d*. *(Value in s designates bit to set.)*
SHL, SHLL SHL *#imm4,Rd* SHLL *#imm4,IM:Rd* SHL *Rs,Rd* SHLL *Rs,IM:Rd*	 B0 B1 B2 B3	 *n*+2 2*n*+2 *n*+3 2*n*+3	* * * *	Shift left register arithmetic: (d) << *n* → (d). (arithmetic left shift — source contains shift count *n*).

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
SHL4 *Rs,Rd*	7A	2	* * – –	Shift left logical 4 bits: *Rs* << 4 → *Rd.*
SHL8 *Rs,Rd*	7B	2	* * – –	Shift left logical 8 bits: *Rs* << 8 → *Rd.*
SHR8 *Rs,Rd*	7C	2	* 0 – –	Shift right 8 bits: *Rs* >> 8 → *Rd.*
STBIT, STBITB STBIT *#imm4,Rd* STBITB *#imm4,*disp16[Rd]* STBIT *Rs,Rd* STBITB *Rs,*disp16[Rd]* *variants:* STBITB *#imm4,&address* STBITB *Rs,&address*	96 97 E6 E7 97 E7	2 6 3 7 6 7	* – – –	Store bit in ST, set to carry value: ~(bit in *d*) → (Z[[ST]]) (C[[ST]]) → (bit in *d*). *(s designates which bit in d.)*
STEA **disp16[Rs],*Rd+* *variant:* *&address,*Rd+*	F1 F1	3 3	– – – –	Store effective address: *disp16 + (Rs)* → *(Rd)* *(Rd) + 2* → *(Rd).*
STRI *#imm4,Rd*	A9	2	0 0 0 0	Store ST, set interrupt level: (ST) → *(Rd).* *imm4* → bits L2–L0 of ST 0s → bits Z, N, C, V of ST
SUB, SUBB *Rs,Rd* *Rs,*disp16[Rd]* *#imm16,Rd* **disp16[Rs],Rd* *variants* *Rs,&address* *&address,Rd*	39 38 3B 3A 3D 3C 3F 3E 3B 3A 3F 3E	1 5 2 3 5 3	* * * *	Subtract source from destination: *(d) – (s)* → *(d).*
SUBQ, SUBQB *#imm4,Rd* *#imm4,*disp16[Rd]* *variant* *#imm4,&address*	87 86 89 88 89 88	1 5 	* * * *	Subtract quick immediate value from dest: *(d) – imm4* → *(d).*
SUBR, SUBRB *RA,RB*	2F 2E	1	* * * *	Subtract with reverse destination: *(RB)* – *(RA)* → *(RA).*
SWAPB *Rs,Rd*	FD	3	* * – 0	Swap bytes, *Rs* to *Rd:* *Rs (LSbyte)* → *Rd (MSbyte)* *Rs (MSbyte)* → *Rd (LSbyte)*

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)

Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
TBIT0 *#imm8,&addr*	F4	3	* – – –	Test for multiple bits clear: IF *imm8* ≠ 0 and *imm8* & *addr* = 0, THEN 1 → (Z[[ST]]) ELSE 0 → (Z[[ST]]) *(test for bit(s) cleared in d;* *s = mask specifying bits to check.)*
TBIT1 *#imm8,&addr*	F5	3	* – – –	Test for multiple bits set: IF *imm8* ≠ 0 and *imm16* & (~*addr*) = 0 THEN 1 → (Z[[ST]]) ELSE 0 → (Z[[ST]]) *(test for bit(s) set in d;* *s = mask specifying bits to check.)*
TBLU, TBLUB TBLUB *Rs,IM,:Rd* *Value 1 ≤ Value 2* *Value 1 > Value 2* TBLU *Rs,IM:Rd* *Value 1 ≤ Value 2* *Value 1 > Value 2*	7F 7E 	 14 15 15 16	* * 0 0	Look up two consecutive values in a table of unsigned data; perform a rounded straight-line interpolation between the two values according to an interpolation fraction.
TRAP *imm8*	FF	7	– – – –	(ST) → ((SP)) (SP) + 2 → (SP) Next inst. addr → ((SP)) (SP) + 2 → (SP) 2 × ~*enumerator8*+ *trap_base_addr* → (PC) 1112 → (ST bits L2–L0). *imm8* value = trap number; ones-complement of trap number becomes *enumerator8* which resides in LSbyte of opcode. *trap_base_addr* = base address of interrupt traps.
TRUNCS, TRUNCSL TRUNCS *Rd* *bits 15–7 equal* *bits 15–7 not equal* TRUNCSL *IM:Rd*	AE AF	 3 4 4	* * 0 *	Test whether signed data can be truncated (represented in next smaller size — word or byte). If not possible, 1 → (V[[ST]]).
TRUNCU *Rd*	AD	2	* * * 0	Test whether an unsigned *word* can be truncated and represented as a *byte* value. If not possible, 1 → (C[[ST]]).

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)

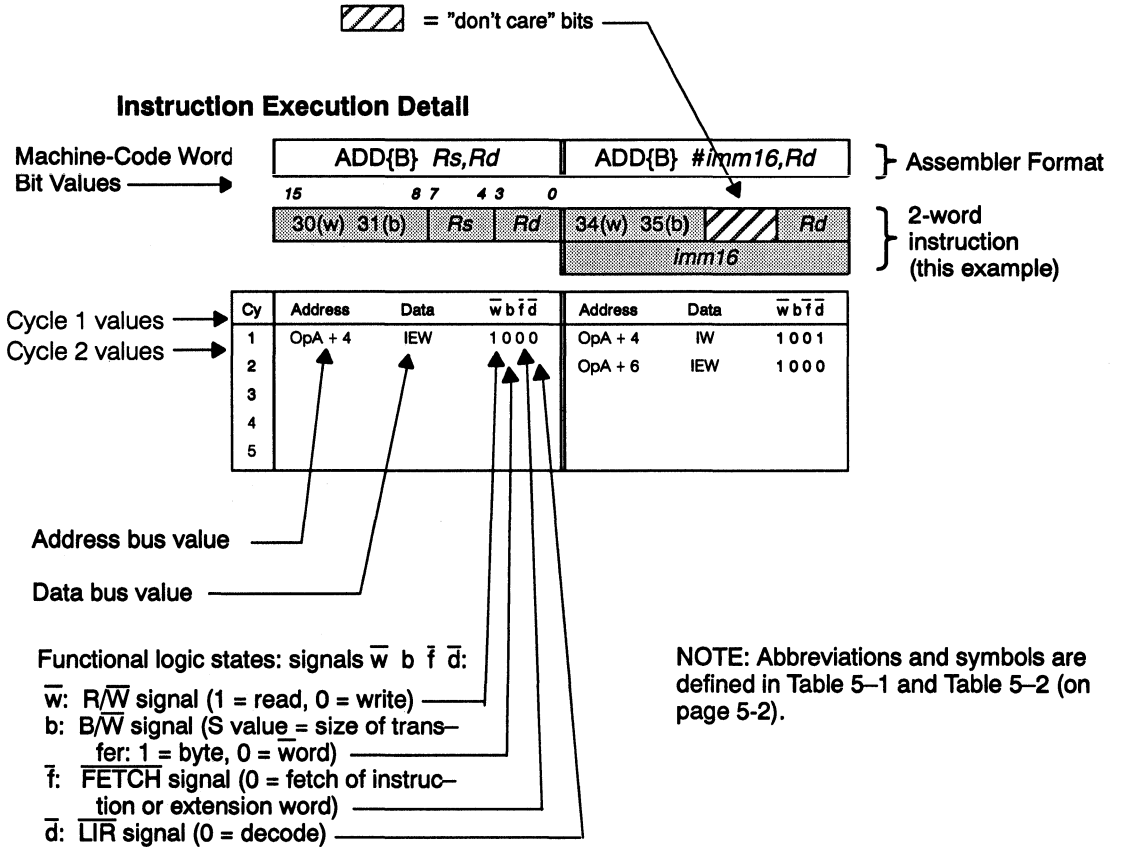
Mnemonic	Opcode† B W L	Cycles (t_c)	Status‡ Z N C V	Operation Description
TST, TSTB¶			* * – 0	Test source:
Rs	03 02	1		*(s)* → (ZR)
**Rs*	0B 0A	2		set Z[[ST]] and N[[ST]] accordingly.
**Rs+*	13 12	3		
#imm16	1B 1A	2		Synthesized as MOV *s,ZR.*
**disp16[Rs]*	23 22	3		
**–Rs*	2B 2A	3		
variant:				
&address	23 22	3		
UNLINK	F6	3	– – – –	Unlink and deallocate stack frame: (FP) → (SP) ((SP)) → (FP).
XNOR, XNORB			* * – 0	Exclusive NOR source with destination:
Rs,Rd	2D 2C	1		*~(s ^ d)* → *(Rd).*
XOR, XORB			* * – 0	Exclusive OR source with destination:
Rs,Rd	51 50	1		*(s) ^ (d)* → *(d).*
*Rs,*disp16[Rd]*	53 52	5		
#imm16,Rd	55 54	2		
*#imm16,*disp16[Rd]*	57 56	5		
variants:				
Rs,&address	53 52	5		
#imm16,&address	57 56	5		

Legend: † Data Size: B = affects byte W = affects word L = affects long word
‡ Status Register Values:
0 = status bit always cleared 1 = status bit always set
– = status bit unchanged by execution * = other effect on status bit (see instruction description)
¶ Synthetic instruction

5.3 Instruction Descriptions in Alphabetical Order

This section contains detailed descriptions of each TMS370C16 instruction, Including bus and signal-line content during each cycle. Variants on an instruction are not covered in this section, but are noted throughout the table in Section 5.2 (starting on page 5-4) and explained in the paragraph on page 5-4.

Figure 5–1. Interpreting the Instruction Execution Detail



Note: Assembler Statements Are Not Case Sensitive

TMS370C16 assembly language statements are not case sensitive. You can enter them in lowercase, uppercase, or a combination. To emphasize this, assembly language statements are shown throughout this user's guide in both uppercase and lowercase.

Syntax **ADC**

Execution (source16) + (destination16) + carry bit → (destination16)

Modes Supported *Rs,Rd*
**disp16[Rs],Rd*

Status Bits

Z	cleared if the result is nonzero; unchanged otherwise
N	equals bit 15 of the result
C	set if an unsigned overflow occurred; cleared otherwise
V	set if a twos-complement overflow occurred, cleared otherwise

Description Add the contents of the source operand and the value of the carry bit of the status register to the destination-register contents (sum remains in the destination register). Source and destination are 16-bit words.

The operation facilitates 32-bit addition. Use an ADD instruction to add the least significant words; then follow with an ADC instruction, adding the most significant words as well as the carry-bit value (the C[[ST]] = 1 if the just-executed ADD instruction included a carry). Thus, the ADD and ADC instructions **must be sequential.**

The Z[[ST]] bit correctly reflects the result of 32-bit addition. The bit is set *only if* the previous operation (like the ADD instruction) set it. Thus, all status bits reflect a 32-bit result *after* an ADD/ADC sequence.

Examples

```
LABEL     ADC  zr,r11          ; Add contents of ZR, R11,
                                ;  and carry bit. Store sum
                                ;  in R11. Effectively a
                                ;  continuous increment of
                                ;  R11 depending on carry
                                ;  bit contents.
LOAD_BUF  ADC  *1000h[r6],r7   ; Add contents at (R6) +
                                ;  1000h plus carry-bit
                                ;  value to R7 contents.
                                ;  Result to R7.
```

Instruction Execution Detail

ADC *Rs,Rd*			ADC **disp16[Rs],Rd*		
8A(w)	*Rs*	*Rd*	8B(w)	*Rs*	*Rd*
			disp16		

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d	Address	Data	$\overline{w}$ b $\overline{f}$ d
1	OpA + 4	IEW	1000	OpA + 4	IW	1001
2				disp + Rs	(disp + Rs)	1011
3				OpA + 6	IEW	1000

Syntax **ADD{B}**

Execution (source) + (destination) → (destination)

Modes Supported
Rs,Rd
**disp16[Rs],Rd*
*Rs,*disp16[Rd]*
#imm16,Rd

Status Bits
Z set if the result is zero, cleared otherwise
N equals bit 7 of the result (byte) or bit 15 of the result (word operation)
C set if an unsigned overflow occurred; cleared otherwise
V set if a twos-complement overflow occurred, cleared otherwise

Description Add the contents of the source to the contents of the destination.

For **byte operations**, sign extend the byte operands to word length, then operate on the word to produce a word result. The most significant byte of the result becomes either 00h for C[[ST]] = 0, or 01h for C[[ST]] = 1. Registers receive the entire word; nonregister destinations receive the least significant byte of the result.

Status bits are set with respect to the size (byte/word) of the operation requested.

Examples

```
LABEL   ADD   R5,R10          ; Add the contents of R5 &
                              ;  R10; store sum in R10.
                              ;
        ADD   *201h[ZR],R12   ; Add contents of location
                              ;  201h and ZR to contents
                              ;  of R12, store sum in R12.
                              ;
        ADDB  *10[r8],r9      ; Add byte contents at 10 +
                              ;  (R8) to R9. Sum goes to
                              ;  LSbyte of R9 with MSbyte
                              ;  of R9 zeroed out.
                              ;
        ADD   #BUFFER,r11     ; Add immediate value of
                              ;  BUFFER and R11. Store
                              ;  results in R11.
```

Instruction Execution Detail

ADD{B} *Rs,Rd*				ADD{B} *#imm16,Rd*			
30(w) 31(b)	*Rs*	*Rd*		34(w) 35(b)	//////	*Rd*	
				imm16			

Cy	Address	Data	$\overline{w}b\overline{f}\overline{d}$	Address	Data	$\overline{w}b\overline{f}\overline{d}$
1	OpA + 4	IEW	1000	OpA + 4	IW	1001
5				Opa+6	IEW	1000

ADD{B} **disp16[Rs],Rd*				ADD{B} *Rs,*disp16[Rd]*			
36(w) 37(b)	*Rs*	*Rd*		32(w) 33(b)	*Rs*	*Rd*	
disp16				*disp16*			

Cy	Address	Data	$\overline{w}b\overline{f}\overline{d}$	Address	Data	$\overline{w}b\overline{f}\overline{d}$
1	OpA + 4	IW	1001	OpA + 4	IW	1001
2	disp + Rs	(disp + Rs)	1S11	disp + Rd	(disp + Rd)	1S11
3	OpA + 6	IEW	1000	prevA	(prevA)	1011
4				disp + Rd	result	0S11
5				OpA + 6	IEW	1000

Note: The $\overline{w}b\overline{f}\overline{d}$ Column Values

Values for the $\overline{w}b\overline{f}\overline{d}$ column are listed in Figure 5–1 on page 5-16.

Syntax **ADQ{B}**

Execution immediate data + (destination) → (destination)

Modes Supported *#imm4,Rd*
*#imm4,*disp16[Rd]*

Status Bits

Z set if the result is zero, cleared otherwise
N equals MSB in result: bit 7 (byte operation) or bit 15 (word operation)
C set if an unsigned overflow occurred; cleared otherwise
V set if a twos-complement overflow occurred; cleared otherwise

Description Add quick immediate data to the contents of the destination operand. (*Quick immediate* data is a 4-bit value contained in the instruction word). The value of 0–15 is zero-extended to a word for addition. ADQ, with its 4-bit immediate operand, operates in only one cycle; whereas, ADD, with a 16-bit immediate operand, uses two cycles.)

For **byte operations**, the byte operands are extended to word length, then operated on as words to produce a word result. The most significant byte of the result will be either 00h when C[ST] = 0 or 01h when C[ST] = 1. Registers receive the entire word, while nonregister destinations receive the least significant byte of the result.

Status bits are set with respect to the size (byte/word) of the operation requested.

Examples

```
LABEL    ADQ  #BITS,R4      ; Add value 'BITS' to R4.
                            ;  Store sum in R4.
ADD_4    ADQ  #4,&BUFFER    ; Add immediate value 4
                            ;  to 'BUFFER'.
```

Instruction Execution Detail

ADQ{B} *#imm4,Rd*			ADQ{B} *#imm4,*disp16[Rd]*		
82(w) 83(b)	*imm*	*Rd*	84(w) 85(b)	*imm*	*Rd*
			disp16		

Cy	Address	Data	w̄ b f d̄	Address	Data	w̄ b f d̄
1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
2				disp + Rd	(disp + Rd)	1 S 1 1
3				OpA + 4	IW	1 0 0 1
4				disp + Rd	result	0 S 1 1
5				OpA + 6	IEW	1 0 0 0

Syntax **AND{B}**

Execution (source) & with (destination) → (destination)

Modes Supported
Rs,Rd
*Rs,*disp16[Rd]*
#imm16,Rd
*#imm16,*disp16[Rd]*

Status Bits
Z set if the result is zero, cleared otherwise
N equals bit in result: bit 7 (byte operation) or bit 15 (word operation)
C unchanged
V cleared

Description Logically AND the contents of the source with the contents of the destination.

For **byte operations**, byte operands are zero-extended to words, operated on words, and produce a word result. The most significant byte of the result will always be 00h. Registers receive the entire word; while nonregister destinations receive the least significant byte of the result.

Status bits are set according to size (byte/word) of the operation.

Examples

```
LABEL     AND   R5,R10          ; AND the contents of R5
                                ;  and R10. Store result
                                ;  in R10.
ready     andb  #clear8,r6      ; AND byte value of CLEAR
                                ;  with R6. Store LSbyte of
                                ;  result in R6, and clear
                                ;  MSbyte of R6.
          AND   #55AAh,R7       ; ADD value of 55AAh with
                                ;  contents of R7. Store
                                ;  result in R7.
```

Instruction Execution Detail

AND{B} *Rs,Rd*				AND{B} *Rs,*disp16[Rd]*			
40(w) 41(b)	*Rs*	*Rd*		42(w) 43(b)	*Rs*	*Rd*	
				disp16			

Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	OpA + 4	IEW	1 0 0 0	OpA + 4	IW	1 0 0 1
2				disp + Rd	(disp + Rd)	1 S 1 1
3				prevA	(prevA)	1 0 1 1
4				disp + Rd	result	0 S 1 1
5				OpA + 6	IEW	1 0 0 0

AND{B} *#imm16,Rd*				AND{B} *#imm16,*disp16[Rd]*			
44(w) 45(b)			*Rd*	46(w) 47(b)			*Rd*
imm16				*disp16*			
				Imm16			

Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	OpA + 4	IW	1 0 0 0	OpA + 4	data	1 0 0 1
2	OpA + 6	IEW	1 0 0 0	disp + Rd	(disp + Rd)	1 S 1 1
3				OpA + 6	IW	1 0 0 1
4				disp + Rd	result	0 S 1 1
5				OpA + 8	IEW	1 0 0 0

Syntax **ASR{L}**

Execution right-shift (destination) by source count → (destination)

Modes Supported

#imm4,Rd	*(word)*
#imm4,IM:Rd	*(longword)*
Rs,Rd	*(word)*
Rs,IM:Rd	*(longword)*

Status Bits

Z set if the result is zero, cleared otherwise

N equals MSB in result: bit 15 of Rd (word operation) or bit 15 of IM (longword operation)

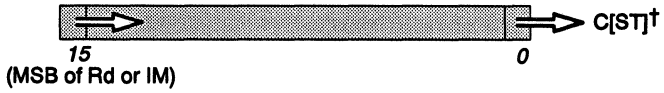
C equals the last bit shifted out of the register; cleared if the shift count in *Rs* is zero

V cleared

Description

Arithmetically right shift the destination register's contents by the number of bit positions (0–15) specified in the source operand. Leave unchanged the preshift value of the most significant bit constant. If the shift count is in a register, the count range (0–15) is defined by the 4 LSBs of the source register (*Rs* bits 15 – 4 are ignored).

The following illustrates a right shift of the most significant bit into the register:

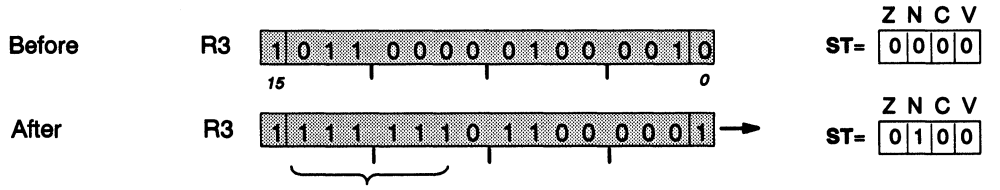


† The value of the last bit shifted out goes to the carry bit; this bit is cleared if the shift count in Rs is zero.

Status bits are set with respect to the size of the word shifted (16 or 32 bits). Longword shifts always use register IM as the most significant word of the 32-bit object. The result of ASR (source),IM:IM is undefined.

Examples

```
LABEL   ASR  #6,R3        ; set R3 to sign bit value
```



Note that if the shift count was changed to 15, R3 would be all ones.

```
Label     ASR    #3,r3        ; Shift R3 three bits right
                              ;
shift     asrl   #2,im:r9     ; Shift the long word in
                              ;  registers IM:R9 right
                              ;  two bits
```

Instruction Execution Detail

ASR *#imm4,Rd*

B4 (w)	*imm4*	*Rd*

Cycle/ Period†	Address	Data	w b f d
n (repeat)	prevA	(prevA)	1 0 1 1
n + 1	OpA + 4	IEW	1 0 0 0

ASR *Rs,Rd*

B6 (w)	*Rs*	*Rd*

Cycle/ Period†	Address	Data	w b f d
1, 2	prevA	(prevA)	1 0 1 1
n (repeat)	prevA	(prevA)	1 0 1 1
n + 1	OpA + 4	IEW	1 0 0 0

Total cycles: *n* + 3

ASRL *#imm4,IM:Rd*

B5 (L)	*imm4*	*Rd*

Cycle/ Period†	Address	Data	w b f d
1	prevA	(prevA)	1 0 1 1
2*n*−2 (repeat)	prevA	(prevA)	1 0 1 1
2*n*	OpA + 4	IEW	1 0 0 0

ASRL *Rs,IM:Rd*

B7 (L)	*Rs*	*Rd*

Cycle/ Period†	Address	Data	w b f d
1, 2, 3	prevA	(prevA)	1 0 1 1
2*n*−2 (repeat)	prevA	(prevA)	1 0 1 1
2*n*	OpA + 4	IEW	1 0 0 0

Total cycles: 2*n* + 2, or 3 if *Rs* = 0h

† A single number represents a *given cycle*. An expression of *n* represents the cycle count *after the previous cycles*, depending on the *n* th number of shifts or repeats. Bus and signal values shown are present during these intervals.

Syntax **ASR0{L}** *(Fourth character is a numerical 0.)*

Execution right shift (destination) by source count → (destination)
IF N[[ST]] = 1 *and* C[[ST]] = 1, THEN (destination) + 1 → (destination)

Modes Supported

Rs,Rd	*(word)*
Rs,IM:Rd	*(longword)*
#imm4,Rd	*(word)*
#imm4,IM:Rd	*(longword)*

Status Bits

Z set if the result is zero, cleared otherwise
N equals the most significant bit of the result
C **ASR0:** cleared
ASR0L: set if an FFFF FFFFh result is rounded (incremented) to 0000 0000h; cleared otherwise
V cleared

Description

Shift the signed (arithmetic) destination register's contents to the right by the number of bit positions in the source operand while holding constant the most significant bit of the destination. Thus, the MSB is extended into the destination a number of times equal to the source value. The shift count of 1–15 is derived from the 4 LSBs of the source contents (*Rs* bits 15 – 4 are ignored).

If, after shifting, the result is negative *and* a 1 was shifted out of the register, the result (destination) is incremented. This performs a mathematically correct signed division by a power of 2 (exponent of 2 = number of bits shifted).

The following illustration depicts a right shift of the most significant bit into the destination register:

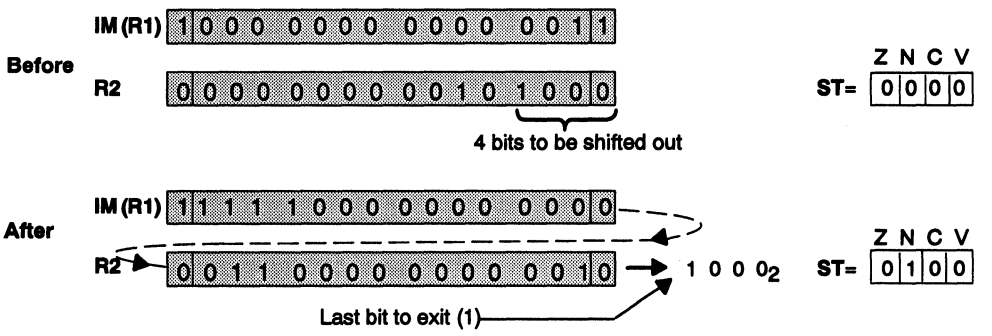
(MSB of Rd or IM)

The shift count is contained in the source (4 LSBs of *Rs* or bits 7–4 of the instruction when *#imm4* is specified). A shift value of 0001_2 to 1111_2 corresponds to a shift of 1 to 15; a shift value of 0000_2 indicates 16.

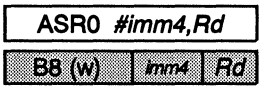
Status bits are set with respect to the size (word/longword) of the operation. Longword shifts always use the IM as the implied most significant word of the 32-bit result. The result of ASR0L (source),IM:IM is undefined.

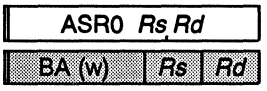
Example

```
ASR0L     #4,IM:R2   ; right shift R1/R2 4 bits
```



Instruction Execution Detail

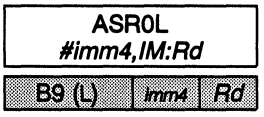


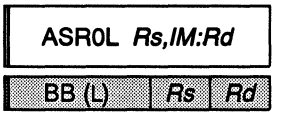


Cycle/ Period †	Address	Data	w b f d
1	prevA	(prevA)	1 0 1 1
n (repeat)	prevA	(prevA)	1 0 1 1
n + 2	OpA + 4	IEW	1 0 0 0

Cycle/ Period†	Address	Data	w b f d
1, 2	prevA	(prevA)	1 0 1 1
3	prevA	(prevA)	1 0 1 1
n (repeat)	prevA	(prevA)	1 0 1 1
n + 2	OpA + 4	IEW	1 0 0 0

Total cycles: *n* + 4





Cycle/ Period †	If N[ST] = 0			If N[ST] = 1		
	Address	Data	w b f d	Address	Data	w b f d
1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2*n* − 1 (repeat)	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2*n* + 1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
2*n* + 2				OpA + 4	IEW	1 0 0 0

Cycle/ Period †	If N[ST] = 0			If N[ST] = 1		
	Address	Data	w b f d	Address	Data	w b f d
1, 2	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
3	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2*n*−1 (repeat)	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2*n* + 1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
2*n* + 2				OpA + 4	IEW	1 0 0 0
	Total cycles: 2*n*+3; or 3 if *Rs* = 0h			Total cycles: 2*n*+4; or 3 if *Rs* = 0h		

† A single number represents a *given cycle*; an expression of *n* represents a *cycle* or *a period of cycles* depending on the n^{th} number of shifts or repeats. Bus and signal values shown are present during these intervals.

Syntax

B{COND}
(where {COND} = condition option; see below)

Execution

If condition is true: (PC) + displacement → (PC)
(where PC = (BCOND_OpA + 4) ÷ 2)
If condition is not true: continue at next instruction in succession

Mode Supported

<displacement8>

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Options

Mnemonic B{COND}	Condition for Branch	Mnemonic B{COND}	Condition for Branch
BC	Carry Set	BLT	Less Than†
BEQ	Equal or Zero	BN	Negative (Minus)†
BGE	Greater Than or Equal†	BNC	Carry Is Clear
BGT	Greater Than†	BNE	Not Equal or Not Zero
BHI	Higher	BNV	Overflow Is Clear†
BHS	Higher or the Same	BP	Positive†
BLE	Less Than or Equal†	BPZ	Plus (Not Negative)†
BLO	Lower	BR	Branch always (no condition)
BLS	Lower or the Same	BV	Overflow Is Set†

† Signed operations (others are logical operations)

Description

If the condition (in ST) is true (one), branch to the address specified. If the condition is not true, go to the next instruction in succession. Table 5–3 explains the conditions for each branch.

The following explains the instruction's branch mechanics, considering the effect of the prefetch pipeline. A maximum *signed* displacement of +127 and –128 words (+254/–256 bytes) can be indicated in the 8-bit signed displacement opcode field. However, this displacement value is figured from the PC value, which points two words past the16-bit word address of the BCOND instruction. This is graphically illustrated in Figure 5–2 (page 5-29) and explained below.

When viewed from the 16-bit PC value, displacement can be figured as +129 words (forward) or –126 words (backward) from the location of the instruction. Actually, a +127 or –128 value (translatable to *words* in displacement) is added to the PC value *when the displacement is figured.* Multiply this sum by 2 to determine the 17-bit BCOND_OpA address. See Figure 5–2 (page 5-29).

To derive the 16-bit PC word address value from the 17-bit BCOND_OpA address, add 4 (the additional 4 bytes beyond the currently executing opcode) and divide by 2. Two methods of destination address calculations:

- starting with the 17-bit memory bus address:
  destination address = BCOND_OpA$_{17}$ + 4 + (disp8_in_bytes × 2)
- starting with the 16-bit PC word value:
  destination address = (PC + disp8_in_words) × 2

  where PC = (BCOND_OpA + 4) ÷ 2.

When a branch is *not* taken (condition false), a clock cycle is saved because the prefetch pipeline does not need to be completely refilled.

Table 5–3. Branches Listed by Opcode

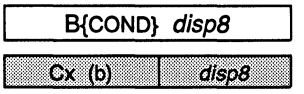
	Mnemonic	Opcode	Description	ST Condition for Branch
	BR	C0h	Branch (unconditional, always)	
Logical Operations	BNC	C1h	Branch if carry clear	C = 0
	BHS	C1h	Branch if higher or the same	C = 0
	BC	C2h	Branch if carry set	C = 1
	BLO	C2h	Branch if lower	C = 1
	BEQ	C3h	Branch if equal or zero	Z = 1
	BNE	C4h	Branch if not equal or not zero	Z = 0
	BHI	C5h	Branch if higher	C | Z = 0
	BLS	C6h	Branch if lower or the same	C | Z = 1
Signed Operations	BGT	C7h	Branch if greater than	Z | (N ^ V) = 0
	BLE	C8h	Branch if less than or equal	Z | (N ^ V) = 1
	BGE	C9h	Branch if greater than or equal	N ^ V = 0
	BLT	CAh	Branch if less than	N ^ V = 1
	BV	CBh	Branch if overflow set	V = 1
	BNV	CCh	Branch if overflow clear	V = 0
	BP	CDh	Branch if positive	N | Z = 0
	BPZ	CEh	Branch if plus (not negative)	N = 0
	BN	CFh	Branch if negative (minus)	N = 1

Note: ^ = XOR, | = OR

Example

```
LABEL     MOV     *R4+,R2   ; Bring in value to R2
          CMP     R2,R3     ; Compare values
          BNE     FAIL_MSG  ; If not = R3, send fail message
          BHI     LABEL     ; If higher, go back 3 words and
                            ; get next value
                  •
                  •
                  •
FAIL_MSG  MOV     R2,*R7    ; Store value
```

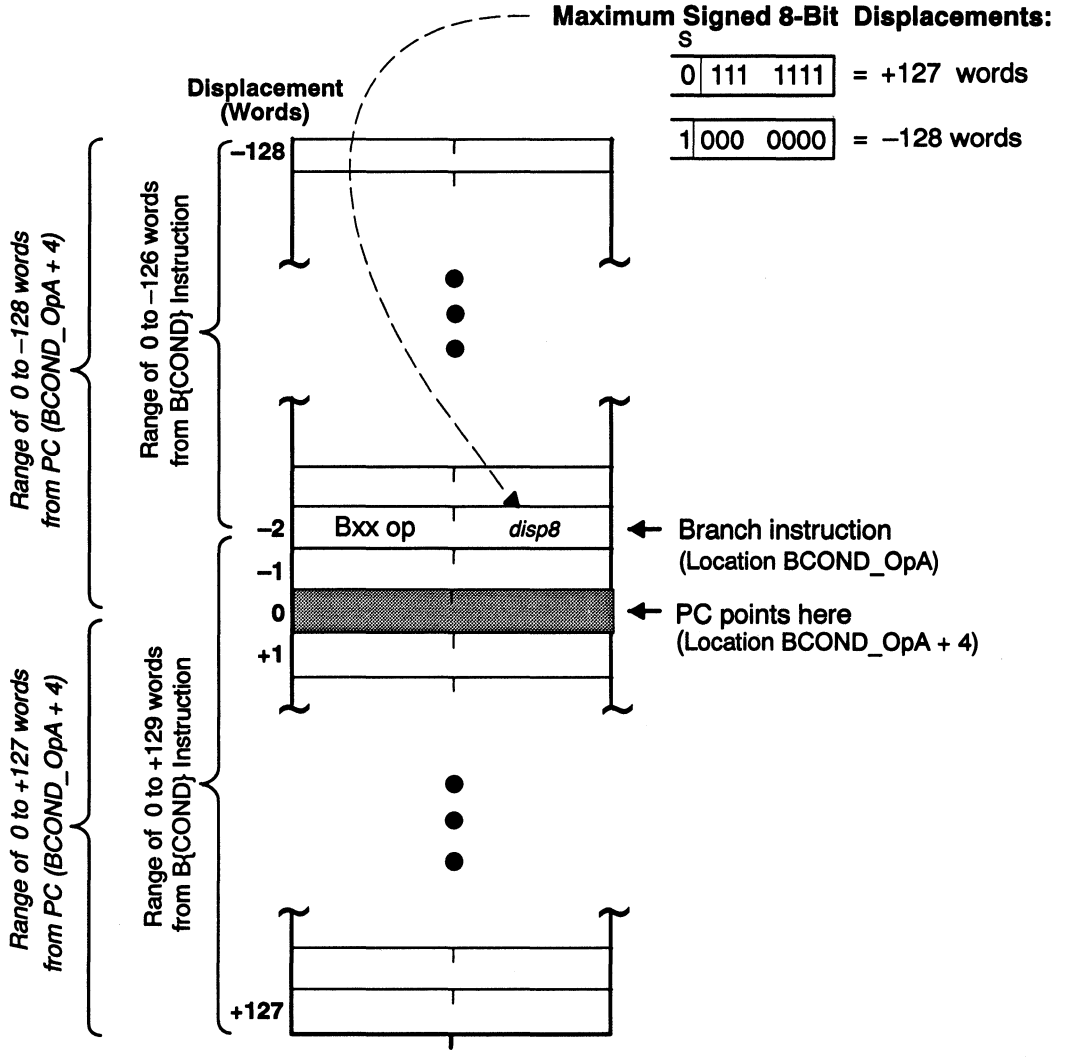
Instruction Execution Detail



Cy	Branch Not Taken			Branch Taken		
	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2	OpA + 4	IEW	1 0 0 0	OpA + 4 +2disp	IW	1 0 0 1
3				OpA + 6 + 2disp	IEW	1 0 0 0

Note: For definitions, see Figure 5–1 on page 5-16.

Figure 5–2. B{COND} Instruction Displacements



Syntax **BRBIT0** *(Last character is a numerical 0.)*

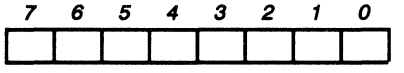
Execution IF bit number *imm3* at byte *addr* = 0, *then* (PC) + *disp8* → (PC)
where PC = (BRBIT0 OpA + 6) ÷ 2] †
ELSE go to next instruction

Mode Supported *#imm3,&addr,disp8* (where: *#imm3* is a number from 0–7, identifying the bit position)

Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description Test a bit (*imm3* = bit number) at a *byte* destination address (*addr*). If **bit = 0**, branch to the specified location by adding the displacement to the PC (add *byte* value — see Figure 5–3 for details). If **bit = 1**, continue to the next instruction following the BRBIT0. If no branch is taken, a clock cycle is saved because the prefetch pipeline does not have to be completely refilled. The destination value addresses only the *first 64K bytes* of memory (address line A0 = 0).

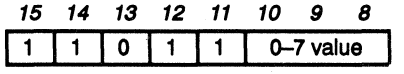
The bit syntax field **must be** in the range 0–7. It is located in the opcode byte (bits 10–8) specifying which bit to test in *addr16* (the byte address). The *imm3* bit value identifies the byte bit according to the following format:



= Bit numbering of destination

The instruction accesses *bytes only,* and it branches *only* if the bit tested is 0.

The *imm3* value is assembled into the three least significant bits of the opcode. This variable value accounts for the D8h–DFh opcode value that specifies the bit number checked in the destination. Opcode format:



= Bit values of BRBIT0 opcode

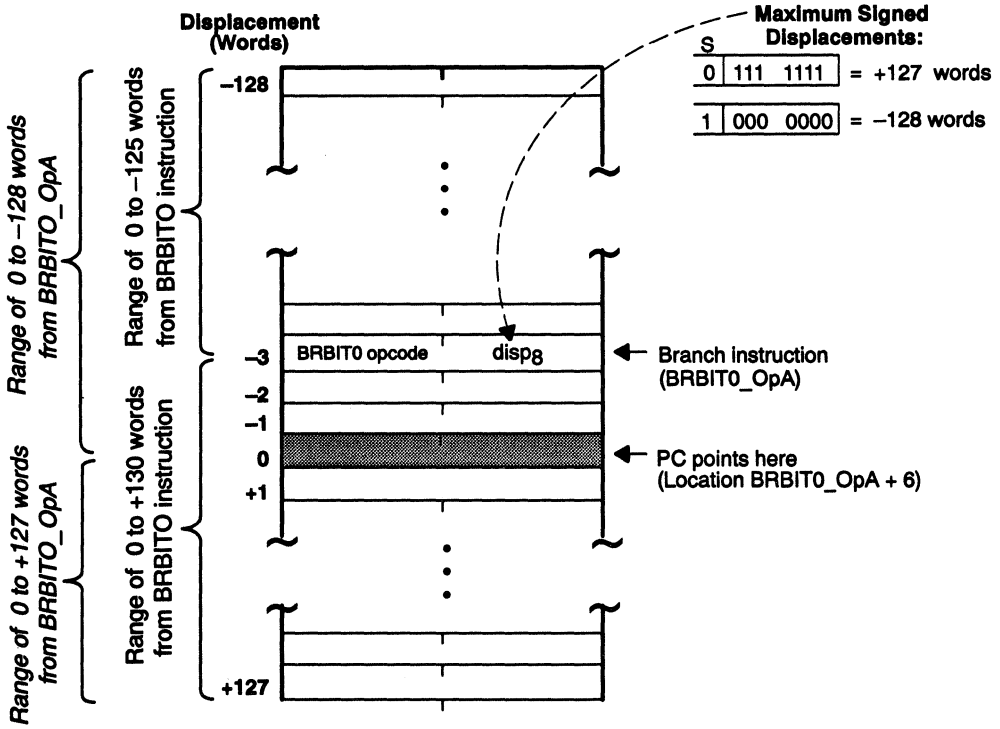
Because the instruction optimally prefetches another word into the pipeline before calculating the destination address, execution flow can be redirected (branched to) by +130 words or –125 words (+260/–250 bytes) as shown in Figure 5–3. This is similar to the BCOND instructions, except that the PC is pointing *six* bytes from the address of the BRBIT0 instruction (instead of *four* bytes from the address of BCOND). (Compare Figure 5–2 and Figure 5–3.)

Example Check the most significant bit at byte address 201. If a 0, go to location TEST; otherwise, continue at the next instruction:

```
Label    BRBIT0      7,&0201,TEST
```

†In the **Execution** entry at the top of the page, the 6 in the OpA + 6 address value is larger than that used for the BCOND or DBNZ instructions because this instruction optimally prefetches another word into the pipeline before calculating a destination address.

Figure 5–3. BRBIT0 and BRBIT1 Instruction Displacements



Instruction Execution Detail

BRBIT0 *#imm3,&addr,disp8*		
11011_2	*imm3*	*disp8*
addr		

Cy	Branch Not Taken			Branch Taken		
	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	addr	(addr)	1 1 1 1	addr	(addr)	1 1 1 1
2	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
3	prev	(prev)	1 0 1 1	prev	(prev)	1 0 1 1
4	OpA + 6	IEW	1 0 0 0	OpA + 4 + (2 x disp)	IW	1 0 0 1
5				OpA + 6 + (2 x disp)	IEW	1 0 0 0

Opcode and Destination Bit to Check

Opcode (hex)	Dest. Bit To Check
D8	0
D9	1
DA	2
DB	3
DC	4
DD	5
DE	6
DF	7

Note: The immediate value designating the bit to set is contained in the three least significant bits of the opcode's left-hand byte. Values are shown in the table on the right.

Syntax **BRBIT1**

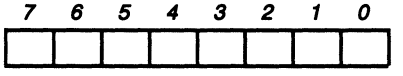
Execution IF bit number *imm3* in address *addr* = 1,
THEN (PC) + *disp8* → (PC)
[where PC = (BRBIT1 OpA + 6) ÷ 2] †
ELSE go to next instruction

Mode Supported *#imm3,&addr,disp8* (where *#imm3* is a number from 0–7, identifying the bit position)

Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description Test a bit (*imm3* = bit number) in the *byte* destination address (*addr*). If **bit = 1**, branch to the specified location by adding the displacement to the PC (add *byte* value — see Figure 5–3 for details). If the **bit = 0**, continue to the next instruction after BRBIT1. If no branch is taken, a clock cycle is saved because the prefetch pipeline does not have to be completely refilled. The destination value addresses only the *first 64K bytes* of memory (address line A0 = 0).

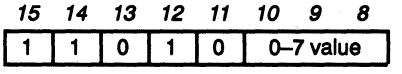
The bit syntax field is a 0–7 value in bits 0–2 of the opcode byte specifying which bit to test at *addr16* (byte address). The *imm3* bit value identifies the byte bit according to the following format:



= Bit numbering of destination

The instruction accesses *bytes only,* and it branches *only* if the bit tested is a 1.

The *imm3* value is assembled into the three least significant bits of the opcode. This variable value accounts for the D0h–D7h opcode value that specifies the bit number checked in the destination. Opcode format:



= Bit values of BRBIT1 opcode

Because the instruction optimally prefetches another word into the pipeline before calculating the destination address, execution flow can be redirected to (branched to) a maximum distance of +130 words or –125 words (+260/–250 bytes) as shown in Figure 5–3. This is similar to the BRBIT0 instruction, except that the branch occurs if the bit is *set.* (The BRBIT0 instruction explanation immediately precedes these pages.)

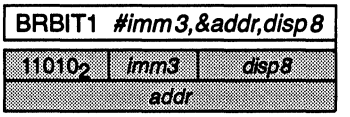
† In the **Execution** entry at the top of the page, the 6 in OpA + 6 value is larger than that used for the BCOND or DBNZ instructions because this instruction optimally prefetches another word into the pipeline before calculating a destination address.

Example

Check the least significant bit (0) in byte address 100. If it is a 1, go to location RECOUNT; otherwise, continue at the next instruction:

```
LABEL    brbit1        0,&100,RECOUNT
```

Instruction Execution Detail



Cy	Branch Not Taken			Branch Taken		
	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	addr	(addr)	1 1 1 1	addr	(addr)	1 1 1 1
2	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
3	prev	(prev)	1 0 1 1	prev	(prev)	1 0 1 1
4	OpA + 6	IEW	1 0 0 0	OpA + 4 + (2 x disp)	IW	1 0 0 1
5				OpA + 6 + (2 x disp)	IEW	1 0 0 0

Note: The immediate value designating the bit to set is contained in the three least significant bits of the opcode's left-hand byte, as shown in the table below.

Opcode and Destination Bit to Check

Opcode (Hex)	Dest. Bit to Check
D0	0
D1	1
D2	2
D3	3
D4	4
D5	5
D6	6
D7	7

Syntax **CALL**

Execution

CALL *addr* or **CALL *Rd:***
(PC) – 2 → ((SP))
(SP) + 2 → (SP)
(destination) → (PC)

CALL **disp16[Rd]* or **CALL **Rd:***
(PC) → ((SP))
(SP) + 2 → (SP)
(disp + (Rd)) → (PC)

Modes Supported

Rd
**Rd* (assembles same as CALL **0000h[Rd]*)
addr
**disp16[Rd]*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description

Jump to the subroutine pointed to by the destination operand. Provide linkage back to the next instruction after CALL by pushing the 16-bit *word address* (PC contents) of the next executable instruction onto the system stack. This return-linkage *word address* (explained in note below) is derived from the *memory* opcode address (OpA) by these equations:

- (CALL_OpA + 2) ÷ 2 for the 16-bit word address of CALL *Rd*
- (CALL_OpA + 4) ÷ 2 for the 16-bit word addresses of CALL *addr16*, CALL **disp[Rd]*, and CALL **Rd*.

CALL *addr* contains a 16-bit *word address* (see note below) to specify the destination. These 16 bits are applied to bits A16–A1 of the address bus (as if shifted left one bit). Note: ***addr17*** must be an *even value*. See Figure 5–4 and Figure 4–12 on page 4-15.

CALL *Rd* jumps to the subroutine at the *word address* in Rd (i.e., *Rd* contents → PC). (Note that **CALL SP**, **CALL **disp16*[SP]**, and **CALL *SP** *are undefined* because SP is *incremented before* execution.)

CALL **disp16[Rd]* and ****Rd*** use *two levels* of indirection to arrive at the destination (see Figure 4–12, page 4-15):

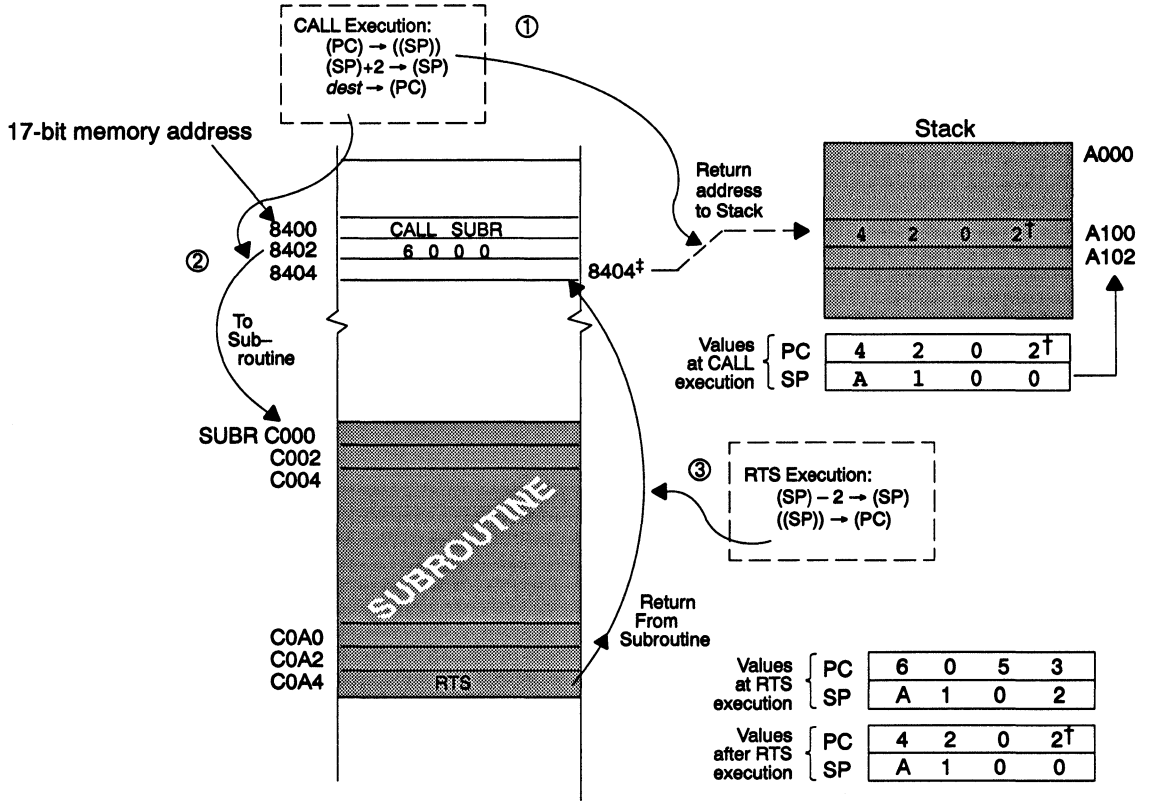
1) Add displacement *disp* and the contents of *Rd* to compute a memory (*not word*) address (*disp* can be 0–FFFFh). This also applies to **CALL **Rd***, which assembles as if written CALL **0h[Rd]*. (If *Rd* is ZR, then *disp16* is the destination address.)
2) At this address, retrieve the *word* address of the destination, which through the PC, is applied to address bus lines A15–A1 with A0 set to 0.

Note: PC's 16-Bit *Word Address* Translates to 17-Bit Address Bus
The program counter's 16-bit *word address* is transformed into a 17-bit physical memory address by overlaying PC data bits 15–0 onto address lines A16–A1 and forcing A0 to 0. See Section 2.3 and Figure 5–4.

Use the RTS instruction to return from the CALL subroutine and continue with the execution of the instruction following the CALL. Use the RTDU instruction to return *if and only if* the subroutine executed a LINK instruction and did not execute an UNLINK instruction.

Example

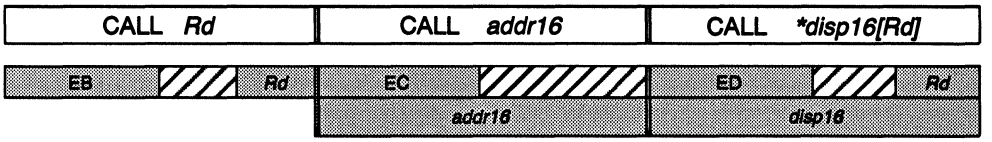
Figure 5–4. CALL and RTS Instruction Example†



† A dashed line denotes the path of the value moved or copied. A solid line denotes a location pointer.

‡The PC value placed on the stack is *one half the 17-bit memory address value.* This is equal to (address of CALL + 4) ÷ 2. On the return, the RTS instruction overlays this stored quotient onto the address bus (essentially multiplying it by 2). This value of one half the address bus value applies to all uses of the PC. This feature is more obvious with addresses above 64K bytes (which require the full 17 address bits).

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	prevA	(prevA)	1 0 1 1	SP	rtnA + 2	0 0 1 1	SP	rtnA + 2	0 0 1 1
2	SP	rtnA + 2	1 0 0 1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
3	prevA	IEW	1 0 0 0	2caddr	IW	1 0 0 1	caddr	disp + Rd	1 0 1 1
4	2Rd	IW	1 0 0 1	2caddr +2	IEW	1 0 0 0	2caddr	IW	1 0 0 1
5	2Rd + 2	IEW	1 0 0 0				2caddr +2	IEW	1 0 0 0

Syntax **CLR{B}** *Synthetic Instruction: Executes as MOV{B} ZR,destination*

Execution (Zero Register) → (destination)

Modes Supported
Rd
**Rd*
**Rd+*
**disp16[Rd]*

Status Bits
Z set
N cleared
C unchanged
V cleared

Description Clear the destination to all zeroes by copying the contents of the R15 (zero register) to the destination.

Register destinations are *completely cleared to 0000h,* even though a byte operation is requested.

Example The following demonstrates various applications:

```
label   CLRB    R12         ; Clear R12 to all zeroes
        CLR     R12         ; Clear R12 to all zeroes
        CLRB    *R11+       ; Clear byte at address
                            ; in R11; increment R11 by 1
        CLR     *R11+       ; Clear contents at address
                            ; in R11; increment R11 by 2
```

Instruction Execution Detail

CLR{B} *Rd*	CLR{B} **Rd*
02(w) 03(b) | 1111_2 | *Rd*	04(w) 05(b) | 1111_2 | *Rd*

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	OpA + 4	IEW	1 0 0 0	Rd	0	0 S 1 1
2				OpA + 4	IEW	1 0 0 0

CLR{B} **Rd+*	CLR{B} **disp16[Rd]*
06(w) 07(b) | 1111_2 | *Rd*	08(w) 09(b) | 1111_2 | *Rd* *disp16*

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	Rd	0	0 S 1 1	OpA + 4	IW	1 0 0 1
2	OpA + 4	IEW	1 0 0 0	disp + Rd	0	0 S 1 1
3				OpA + 6	IEW	1 0 0 0

Syntax **CMP{B}**

Execution compute (destination) – (source); set ST bits according to results

Modes Supported

Rs,Rd	**Rs+,Rd*
#imm16,Rd	**disp1[Rs],*disp2[Rd]*
**disp16[Rs],Rd*	

Status Bits

Z set if result is zero; cleared otherwise
N equals bit in result: bit 7 (byte operation) or bit 15 (word operation)
C set if an unsigned underflow occurred; cleared otherwise
V set if a twos-complement underflow occurred; cleared otherwise

Description

Compare the contents of the source operand to the destination operand and set the ST status bits accordingly.

The *compare is performed* by subtracting the source contents from the destination contents. Results of the operation are reflected in the ST status bits.

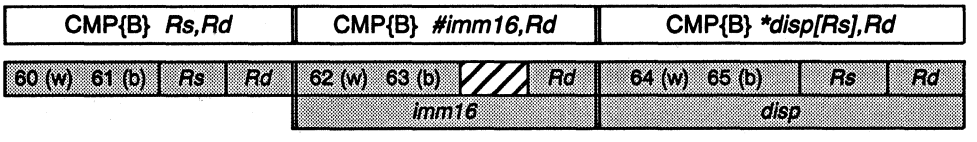
For byte operations, only the least significant bytes of the *register* operands are compared. Status bits are set with respect to the size (byte or word) of the operation.

CMP{B} *Rn+,Rn *is a special-case operand combination where both parts of the operand use the same register. The compare of* ***Rn** *and* **Rn** *occurs* ***before*** Rn *is postincremented.*

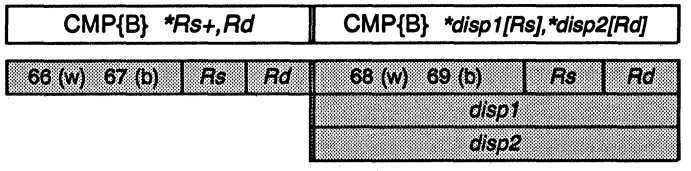
Example

```
LABEL  CMP   R12,R4   ; Is R12 equal to R4?
       BEQ   YES_EQ   ; Yes, go to equal subroutine
       CALL  NOT_EQ   ; No, go to not-equal subroutine
```

Instruction Execution Detail



Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2				OpA + 6	IEW	1 0 0 0	disp + Rs	(disp + Rs)	1 S 1 1
3							OpA + 6	IEW	1 0 0 0



Cy	Address	Data	w b f d	Address	Data	w b f d
1	Rs	(Rs)	1 S 1 1	OpA + 4	disp2	1 0 0 1
2	prevA	(prevA)	1 0 1 1	OpA + 6	IW	1 0 0 1
3	OpA + 4	IEW	1 0 0 0	disp1 + Rs	(disp1 + Rs)	1 S 1 1
4				disp2 + Rd	(disp2 + Rd)	1 S 1 1
5				OpA + 8	IEW	1 0 0 0

Syntax **CMPC**

Execution (destination) – (source) – C[[ST]] bit; set ST codes accordingly

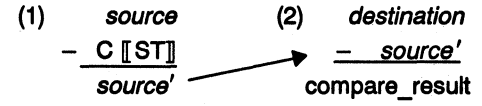
Modes Supported *Rs,Rd*
**disp16[Rs],Rd*

Status Bits

Z	cleared if the result is non-zero; otherwise, unchanged
N	equals the most significant bit of the result
C	set if an unsigned underflow/borrow occurred; otherwise, cleared
V	set if a twos-complement underflow occurred; otherwise, cleared

Description Compare the source value, minus the carry bit value, to the destination. Then set the ST codes according to the comparison.

The comparison is done in the following steps: (1) subtract the carry bit value from the source, *and then* (2) subtract this result from the destination:



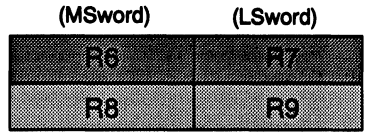
The ST codes reflect the operation, and the result is discarded (source, destination not changed).

This instruction is designed for 32-bit compares with the first words (LSwords) compared using the CMP instruction. The CMPC *immediately follows* the CMP instruction to compare the most significant words. If the CMPC comparison of the MSwords is true, the Z[[ST]] bit remains unchanged, reflecting the earlier comparison of the LSwords by the CMP. However, if an underflow/borrow occurred in the earlier LSword/CMP comparison, this will be included in the subtraction of the two MSwords during the CMPC comparison. Thus, two alike most significant values will show a zero Z[[ST]] bit because of the carry over from the least significant comparison.

Therefore, all ST condition codes will reflect a 32-bit compare after a CMP/CMPC sequence of compares is executed.

Example Compare two 32-bit values — contents of R6/R7 with R8/R9 (two MSword/LSword combinations). If equal, branch to subroutine EQUAL:

```
LABEL   CMP     R7,R9     ; Compare LS words
        CMPC    R6,R8     ; Compare MS words
        BEQ     EQUAL     ; If equal, branch
```



Instruction Execution Detail

CMPC *Rs,Rd*			CMPC **disp16[Rs],Rd*		
8E (w)	*Rs*	*Rd*	8F (w)	*Rs*	*Rd*
			disp16		

Cy	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	OpA + 4	IW	1 0 0 1
2				disp + Rs	(disp + Rs)	1 S 1 1
3				OpA + 6	IEW	1 0 0 0

Syntax **COMPL{B}** *Synthetic Instruction: Executes as SUBR{B} Rn,ZR*

Execution (ZR) – (Rn) (effectively SUBR{B} *Rn,ZR)*
result → (Rn)

Mode Supported *Rn*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C cleared if the result is zero; set otherwise
V set if a twos-complement underflow occurred; cleared otherwise

Description Compute the negative value (perform a twos-complement negation) of the destination register's contents by subtracting its contents from zero and placing the result in the destination register.

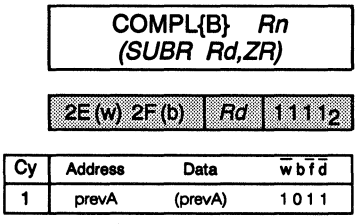
For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The *most significant byte* of the result will be either 00h for C[[ST]]=0 or FFh for C[[ST]]=1.

Status bits are set with respect to the size (byte or word) of the operation.

Examples

```
Label   compl  R11  ; Negate R11 (twos complement)
        COMPLB r4   ; Negate LS byte of R4. MS byte
                    ; value depends on carry bit of
                    ; ST as described above.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,d$
1	prevA	(prevA)	1011

Syntax **DBNZ**

Execution
(*Rs*) – 1 → (*Rs*)
IF *Rs* ≠ 0
THEN (PC – *disp4*) → (PC)
(where PC = [(DBNZ address + 4) ÷ 2]
ELSE go to instruction following DBNZ

Mode Supported *Rs,disp4*

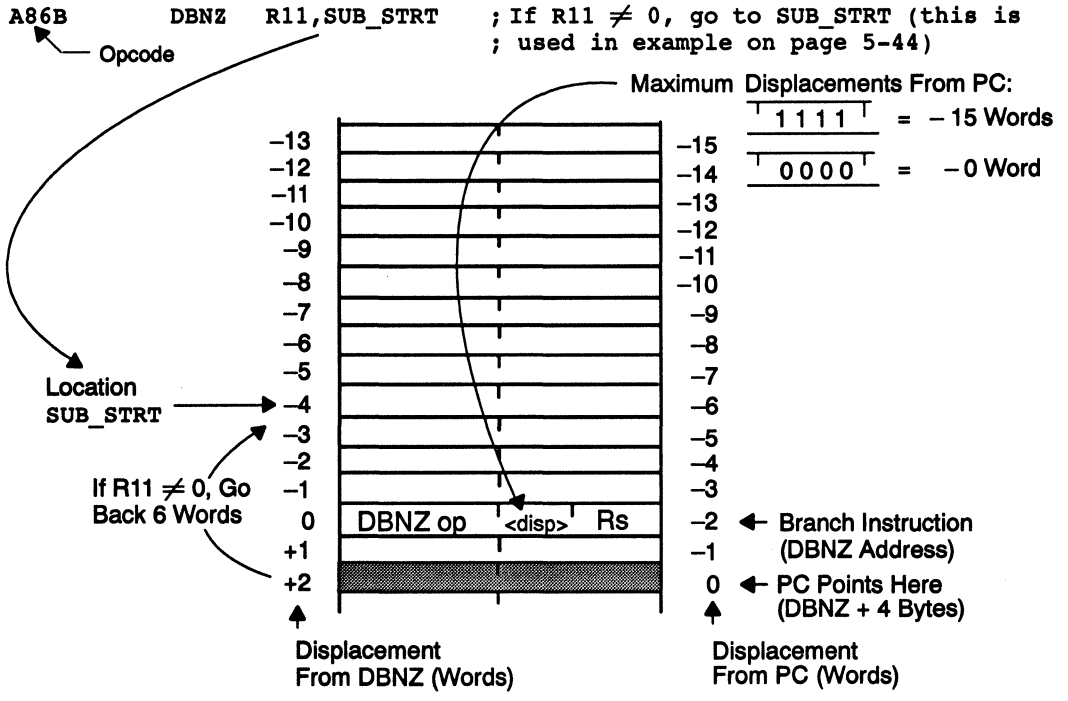
Option **DBNZ** *Rs,<displacement4>*

Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description First, decrement the *Rs* word by one.

Then, If the result is **nonzero**, branch to the location pointed to by *subtracting* the 4-bit displacement from the PC value. Note that this subtraction takes place when the PC is pointing to the four bytes following the DBNZ instruction;

Figure 5–5. DBNZ Displacement Computation



thus, the displacement can redirect execution +2 words (a 0000_2 displacement) or –13 words (1111_2) from the 17-bit address of the DBNZ instruction (see Figure 5–5). *In any case, the branch must be negative — to a previous address (lower memory address).*

But, if the result is **zero**, do not branch; go to the next instruction.

Example This instruction provides a loop counter with the source register containing the number of loops desired. This is graphically shown in the left side of Figure 5–5.

```
            MOV    #100,R11       ; Set up to check 100 bytes
;
;   ******* Start of subroutine ****
;
SUB_STRT    MOVB   *R3+,*R5+      ; Bring in byte (next byte)
                          •
                          •
                          •
;
;      Subroutine manipulates byte, stores it
;
                          •
                          •
                          •
;
;      Now check if 100th byte read
;
            dbnz   R11,SUB_STRT   ; If R11 ≠ 0, go to SUB_STRT,
                                  ; get next byte and repeat;
                                  ; otherwise, exit.
```

Instruction Execution Detail

DBNZ *Rs,disp4*

A8 (w)	*disp4*	*Rs*

	Branch Not Taken			*Branch Taken*		
Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011	prevA	(prevA)	1011
2	OpA + 4 + (2 x disp)	IW	1001	prevA	(prevA)	1011
3	OpA + 6 + (2 x disp)	IEW	1000	prevA	(prevA)	1011
4				OpA + 4	IEW	1000

Syntax **DEC{B}** *Synthetic Instruction: Executes as SUBQ{B} #1,dest*

Execution (destination) – 1 → destination

Modes Supported *Rd*
**disp16[Rd]*

Status Bits
Z set if the result is zero; otherwise, cleared
N equals bit in result: bit 7 (byte operation) or bit 15 (word operation)
C set if an unsigned underflow occurred; otherwise, cleared
V set if a twos-complement underflow occurred; otherwise, cleared

Description Subtract one from the destination register or the destination address. Set the status bits with respect to the byte/word size of the result.

For byte operations, the operand is zero-extended to word size, operated on as a word, and produces a word result. The most significant byte of the result is either:
00h for C[[ST]] = 0 **or**
FFh for C[[ST]] = 1.

Nonregister destinations receive the least significant byte of the result; registers receive the entire word.

Example

```
8611    LABEL   DEC     R1        ; Subtract 1 from R1
8811            DEC     *100[R1]  ; Subtract 1 from value at
0064                              ; address computed as the
                                  ;  sum of R1 contents and
100
```

Instruction Execution Detail

DEC{B} *Rd* *(SUBQ{B} #1,Rd)*				DEC{B} **disp16[Rd]* *(SUBQ{B} #1,*disp16,[Rd])*			
86 (w) 87 (b)	0001_2	*Rd*		88 (w) 89 (b)	0001_2	*Rd*	
				disp16			

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	OpA + 4	IEW	1000	prevA	(prevA)	1011
2				disp + Rd	(disp + Rd)	1S11
3				OpA + 4	IW	1001
4				disp + Rd	result	0S11
5				OpA + 6	IEW	1000

Syntax **DIVS{L}**

Execution
(dest) ÷ (src)
quotient → (Rd)
remainder → (IM)

Modes Supported
Rs,Rd *(word format only — divide 16-bit Rd by 16-bit Rs)*
Rs,IM:Rd *(long format only — divide 32-bit IM:Rd by 16-bit Rs)*

Status Bits

Z if V[[ST]] = 1: bit Z[[ST]] is set if the 16-bit *divisor* is zero; cleared otherwise.
if V[[ST]] = 0: bit Z[[ST]] is set if the 16-bit *quotient* is zero; cleared otherwise.

N equals V[[ST]] bit value XORed with the theoretical sign of the quotient (see last paragraph of **Description**, on page 5-47).

C cleared

V set if a twos-complement overflow of the 16-bit quotient occurs; cleared otherwise (see third paragraph and table in **Description**).

Description

Divide (as signed values) the source register into the destination register(s). Place the quotient in the destination register and the remainder in the implied register (IM). The destination value to be divided is in one or two registers:

- ❑ one for word (16-bit) by 16-bit) division: Rd ÷ Rs
- ❑ two for long (32-bit by 16-bit) division: IM:Rd ÷ Rs (IM and Rd concatenated with **IM the most significant word**)

Note that the sign of the remainder is *the same* as the sign of the original dividend (destination register contents). Also, the result is assigned in the following sequence: the remainder goes to IM first; then, the quotient goes to the destination. Thus, if IM is *also* the Rd in the destination of a long operation (in other words, DIVS Rs,IM:IM), then the remainder in the IM is overwritten by the quotient.

Twos-complement (signed) **overflow** occurs when the quotient does not fit into 16 bits. This occurs under the following conditions:

Operation	Where
DIVS and DIVSL	Rs contains 0000h
DIVS	Rs contains FFFFh (i.e., –1) **and** Rd contains 8000h (i.e., –32,768)
DIVSL	Rs bit 15 = IM bit 15 (|Rs × 32768| ≤ |IM:Rd|) **or** Rs bit 15 ≠ IM bit 15 (|Rs × 32769| ≤ |IM:Rd|)

When such overflows occur, Rs, IM, and Rd will be left unchanged.

The *theoretical sign* of the quotient is the XOR of the most significant bits of the dividend and divisor prior to division. In other words:

- ❑ For DIVS, this is Rs bit 15 XORed with Rd bit 15
- ❑ For DIVSL, this is Rs bit 15 XORed with IM bit 15

Note: Do Not Use Operand Rs,IM:Rs

Using the operand Rs,IM:Rs can produce an undefined result. Depending on the size of the instruction and the contents of IM and Rs, it is possible to get a correct or incorrect result or an overflow.

Examples

```
Label   divs   R8,R9      ; Signed divide of R9 by R8.
                          ;   Result to R9; remainder to IM.
                          ;
LONGL   DIVSL  R8,IM:R2   ; Signed divide of concatenated
                          ;   IM:R2 by R8. Result to R2;
                          ;   remainder to IM.
```

Instruction Execution Detail

DIVS *Rs,Rd*

A2 (w)	*Rs*	*Rd*

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011
2–26	prevA	(prevA)	1011
† final	OpA + 4	IEW	1000

† Word division (DIVS) takes 27 cycles, with the following two exceptions:

Dividend	Divisor	Cycles	Comment
any	0000h	4	overflow
8000h	FFFFh	26	overflow

The last line in the boxed table shows the logic values for the final cycle.

DIVSL *Rs,IM:Rd*

A3 (L)	*Rs*	*Rd*

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011
2–28	prevA	(prevA)	1011
‡ final	OpA + 4	IEW	1000

‡ Longword division (DIVSL) takes 29 cycles, with the following eight exceptions:

Dividend	Divisor	Cycles	Comment
8000 0000h	pos	6	overflow
8000 0000h	neg	7	overflow
$\lvert IM{:}Rd\rvert \geq \lvert Rs\rvert \times 65536$	pos	8	overflow
$\lvert IM{:}Rd\rvert \geq \lvert Rs\rvert \times 65536$	neg	9	overflow
$IM{:}Rd \geq Rs \times 32768$	pos	28	overflow
$IM{:}Rd \leq Rs \times 32768$	neg	28	overflow
$IM{:}Rd \geq -Rs \times 32768$	pos	28	overflow
$IM{:}Rd \leq -Rs \times 32768$	neg	28	overflow

The last line in the boxed table shows the logic values for the final cycle.

Syntax **DIVU{L}**

Execution
(dest) ÷ (src)
quotient → (Rd)
remainder → (IM)

Modes Supported
DIVU *Rs,Rd* ***(word format only** — divide 16-bit Rd by 16-bit Rs)*
DIVUL *Rs,IM:Rd* ***(long format only** — divide 32-bit IM:Rd by 16-bit Rs)*

Status Bits

Z if C[[ST]] = 1: set if the 16-bit divisor is zero; cleared otherwise
if C[[ST]] = 0: set if the 16-bit quotient is zero; cleared otherwise

N if C[[ST]] = 1: equals the most significant bit of the 16-bit divisor
if C[[ST]] = 0: equals the most significant bit of the 16-bit quotient

C set if an unsigned overflow of the 16-bit quotient occurred; cleared otherwise

V cleared

Description

Divide (as unsigned values) the source register into the destination register(s). Place the quotient in the destination register and the remainder in the implied register (IM). The destination value to be divided is in one or two registers:

- ❑ one for word (16-bit by 16-bit) division: Rd ÷ Rs
- ❑ two for long (32-bit by 16-bit) division: IM:Rd ÷ Rs (IM and Rd concatenated, with **IM the most significant word**)

The result assignment sequence is the remainder to IM first and then the quotient to Rd. If Rd is also IM (for example, DIVU Rs,IM:IM), then the remainder in the IM is overwritten by the quotient.

Unsigned overflow occurs when the quotient does not fit in a 16-bit data object. This occurs for the following conditions:

- ❑ DIVU and DIVUL: Rs contains 0000h
- ❑ DIVUL: Rs × 65,536 ≤ IM:Rd (for example, Rs ≤ IM)

When such overflows occur, Rs, IM, and Rd will be left unchanged.

Examples

```
LABEL   DIVU   R2,R3      ; Signed divide of R3 by R2.
                          ;  Result to R3; remainder
                          ;  to IM.
Long2   divul  R4,IM:R2   ; Signed divide of concatenated
                          ;  IM and R2 by R4. Quotient to
                          ;  R2, remainder to IM.
```

Instruction Execution Detail

DIVU *Rs,Rd*			DIVUL *Rs,IM:Rd*		
A0 (w)	*Rs*	*Rd*	A1 (L)	*Rs*	*Rd*

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2–20	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
final †	OpA + 4	IEW	1 0 0 0	OpA + 4	IEW	1 0 0 0

† **Word division (DIVU)** takes 21 cycles unless the divisor is 0000h; in which case, it takes only 5 cycles with an overflow occurring. The third line in the table shows final-cycle logic values. **Longword division (DIVUL)** takes 21 cycles unless the divisor is 0000h **or** unless IM:Rd $\geq$ Rs $\times$ 65536. These two DUVUL exceptions take only 4 cycles with an overflow occurring. The third line in the table shows the final-cycle logic values.

Syntax **EXTS{B}**

Execution

value of (*Rd7*) → (*Rd8–Rd15*) *(byte)*

or

value of *Rd15* → *IM0–IM15* *(word)*

Modes Supported

EXTSB	*Rd*	(for EXTSB only — *byte operation)*
EXTS	*IM:Rd*	(for EXTS only — *word operation)*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C for **EXTS**: equals N⟦ST⟧
for **EXTSB**: cleared
V cleared

Description

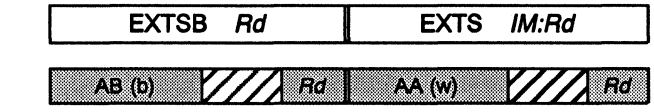
Sign-extend the data in the destination register to the next larger data size (for example, extend byte to word or word to 32-bit longword).

For word to longword, implied register IM is the destination for the most significant word of the 32-bit result.

Examples

```
label   exts    im:r3   ; (sign) extend bit 15 of R3
                        ;  through the IM (R1)
                        ;
        extsb   r3      ; (sign) extend the value of R3's
                        ;  bit 7 through R3's MSbyte
```

Instruction Execution Detail



Cy	Address	Data	w̄ b̄ f̄ d̄	Address	Data	w̄ b̄ f̄ d̄
1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
2				OpA + 4	IEW	1 0 0 0

Syntax **EXTZ{B}** *Synthetic Instruction: Executes as a MOV Instruction*

Execution
(Rd) → *(Rd)* (For EXTZB: same as MOVB *Rd,Rd.*)
(ZR) → *(IM)* (For EXTZ: same as MOV *ZR,IM.*)

Modes Supported
Rd (For EXTZB only: *byte operation.)*
IM:Rd (For EXTZ only: *word operation.* No matter what register is specified, the instruction *always* clears the IM register only.)

Status Bits
Z set if the result is zero; cleared otherwise
N equals bit 7 of Rd for EXTZB; cleared for EXTZ
C unchanged
V cleared

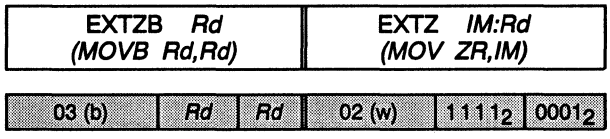
Description
Extend the unsigned data in the destination register to the next larger data size. This extends *byte to word* by zeroing the destination register's most significant byte and extends *word to longword* by clearing the concatenated IM register. In other words:

- For the *byte* instruction (EXTZB) execution clears the MSbyte of the destination register.
- For the *word* instruction (EXTZ), execution clears *only* the IM register (R1), no matter which register specified.

Examples

```
Label       EXTZB    R5       ; Clear MSbyte of R5
                                ;
Clear_IM    Ext      IM:r5    ; Clear IM register
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	OpA + 4	IEW	1000	OpA + 4	IEW	1000

Syntax **FMOV**

Execution (source) → (destination)

Modes Supported *Rs,*Rd* *(where *Rs and *Rd mean indirect address)*
**Rs,Rd*

Status Bits
Z set if the transferred data was zero; cleared otherwise
N equals the most significant bit of the transferred data
C unchanged
V cleared

Description This instruction moves data to or from the upper half of the memory space. The *indirection* register **Rn* accesses the full 128K bytes of memory and contains a *word* address. The other register addresses the first 64K bytes of memory and contains the specified memory address.

The content of the indirection register forms a 17-bit physical memory address by overlaying register bits 15–0 onto address lines A16–A1, then forcing A0 to 0. Because the indirection-register contents are one half the address bus value, the example below (and in Section 4.8, page 4-16) illustrates the use of the ? operator to load this address value into the register.

Example Move the contents of R2 into address 1C400h:

```
MOV     #?1C400h,R4    ; place addr 1C400h/2 in R4.
FMOV    R2,*R4         ; move contents of R2 to 1C400h
```

The first instruction places E200h (1C400h ÷ 2) into R4. During the FMOV instruction, the E200h is applied to address bus lines A15–A1 with A0 a 0, deriving the destination address 1C400h. Note that the instruction:

```
MOV    #1C400h/2,R4
```

would perform the same function as MOV #?1C400h,R4.

Instruction Execution Detail

FMOV *Rs,*Rd*			FMOV **Rs,Rd*		
F2 (w)	Rs	Rd	F3 (w)	Rs	Rd

Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
3	2Rd	(2Rd)	1 S 1 1	2Rs	(2Rs)	1 S 1 1
4	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
5	OpA + 4	IEW	1 0 0 0	OpA + 4	IEW	1 0 0 0

Syntax **IDLE**

Execution assert IDLE signal;
while in idle state, do nothing

Modes Supported *Operand not necessary for IDLE*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

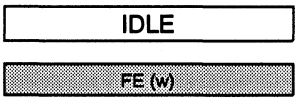
Description Idle the CPU by entering an internal endless "do nothing" loop. The system module then enters the idle, standby, or halt mode (these modes are discussed in Section 3.11 on page 3-40).

Methods of exiting the idle state depend upon device configuration and the idle low-power mode that was entered.

Examples

```
Label    IDLE              ; Enter the idle state or the
                           ;  designated low power mode.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2	prevA	(prevA)	1 0 1 1
while idle	high-z	high-z	high-z

As shown, it takes two cycles to enter the idle state. Once in that state, signals shown at the top of the table (Address, Data, $\overline{w}$, b, $\overline{f}$, $\overline{d}$) remain in the high-impedance state.

Syntax **ILLEGAL**

Execution
(ST) → ((SP))
(SP) + 2 → (SP)
(PC) + 1 → ((SP))
(SP) + 2 → (SP)
((TRAP 0)) → (PC)
ones → L2–L0[[ST]]

Modes Supported *Operand not necessary for ILLEGAL*

Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description Generate a trap exception by pushing the current ST contents and the *word address* of the next executable instruction plus 2 onto the system stack. Then load the PC with the contents of the vector for TRAP 0 (traps are further described in subsection 3.7.6 on page 3-24). It is preferred that the trap 0 vector point to a reset sequence.

An RTI instruction returns execution to the interrupted execution flow.

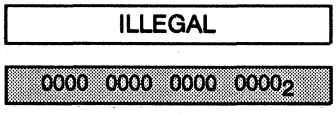
While ILLEGAL has an explicit opcode of 0000h, the following opcodes will generate the same result and are also considered illegal:

6Ah through 6Fh	70h through 79h	81h
98h through 9Fh	EEh and EFh	

Example

```
Label    Illegal      ; Load the PC with the Trap 0
                      ; 'illegal' vector, usually placed
                      ;  in code somewhere that should
                      ;  probably not be used during
                      ;  normal operation. It is suggested
                      ;  to have the Trap 0 routine contain
                      ;  a reset sequence.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b f $\overline{d}$
1	OpA + 4	IEW	1001
2	SP	SR	0011
3	SP + 2	(rtnA + 4) ÷ 2	0011
4	prevA	(prevA)	1011
5	Trap 0 addr	(Trap 0 addr)	1011
6	Trap 0 vector × 2	IW	1001
7	(Trap 0 vector × 2) + 2	IEW	1000

Syntax **INC{B}** *Synthetic Instruction: Executes as ADQ{B} #1,dest (destination + 1 → destination)*

Execution ADQ{B} *#1,dest*

Modes Supported *Rd*
**disp16[Rd]*

Status Bits
Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C set if an unsigned underflow occurred; cleared otherwise
V set if a twos-complement underflow occurred; cleared otherwise

Description Add one to the destination operand. Status bits are set with respect to the size (byte or word) of the operation.

For byte operations:
- Bit C[[ST]] = **0** when the MSbyte is 00h,
  = **1** when the MSbyte is 01h.
- Byte operands are zero-extended to words, are operated on as words, and produce a word result.
- Nonregister destinations receive the least significant byte of the result, while registers receive the entire word.

For word operations, bit C[[ST]] = **1** when the destination increments from FFFFh to 0000h.

Example

```
INCB    R7          ; Increase contents of register 7
                    ;   by 1
INCB    *101h[ZR]   ; Increase the contents of byte
                    ;  address 101h by 1 (ZR = 0)
```

Instruction Execution Detail

INC{B} *Rd* (ADQ{B} #1,Rd)				INC{B} **disp16[Rd]* (ADQ{B} #1,*disp16[Rd])			
82 (w)	83 (b)	0001_2	*Rd*	84 (w)	85 (b)	0001_2	*Rd*
				disp16			

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	OpA + 4	IEW	1000	prevA	(prevA)	1011
2				disp + Rd	(disp + Rd)	1S11
3				OpA + 4	IW	1001
4				disp + Rd	result	0S11
5				OpA + 6	IEW	1000

Syntax

INTPU

Execution

```
IF  (IM) > (Rd)
 THEN
   LSbyte of Rs x (IM – Rd) + 80h → temp      (8 bits x 16 bits → 24 bits + 80h)
   temp ÷ 256 → Rd
   (IM) – (Rd) → (Rd)
ELSE
   LSbyte of Rs x (Rd – IM) + 80h → temp      (8 bits x 16 bits → 24 bits + 80h)
   temp ÷ 256 → Rd
   (IM) + (Rd) → (Rd)
```

Mode Supported

Rs,IM:Rd

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C cleared
V cleared

Description

Perform a rounded straight line interpolation between the values contained in registers IM and Rd according to the interpolation fraction in Rs. (Note that a *colon* (:) separates IM and Rd in the destination's syntax shown in the *Mode Supported* section.)

The interpolation fraction is held in the least significant byte of Rs and has its radix point between bits 7 and 8. The most significant byte of Rs is ignored, and the contents of Rs are left unchanged.

The contents of IM and Rd are treated as words, with all arithmetic operations being word size. Bytes can be used in these registers if the register's most significant byte is zero.

The internal multiply is 8 x 16 bits, where the 8-bit value is the fraction and the 16-bit value is the difference between IM and Rd. The product is a fixed-point value with the integer portion in bits 8–23 and the fraction in bits 0–7. Round up the product to word size by adding 80h, yielding a word value in bits 8–23. The most significant word of the rounded product is then combined with IM, yielding the final interpolated result, which is then placed in Rd.

The fractional portion of the temporary result is lost. The operand combination:

```
INTPU  Rs,IM:IM
```

will always generate a result of 0000h in IM.

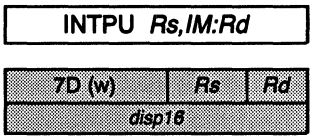
Status bits are set with respect to a word result in Rd.

Example

This example performs a rounded interpolation between the 0000h value in the IM register and 1000h in R6. The interpolation fraction of 256/2 is contained in R5. The result goes to R6, the destination register.

```
        MOV     ZR,IM          ; 0000h to the IM register
        MOV     #1000h,R6      ; 1000h to R6
        MOV     #(256/2),R5    ; Interpolation fraction to R5
LABEL   INTPU   R5,IR:R6       ; Interpolate between values in
                               ;  in IM and R6, with rounding;
                               ;  result is in R6
```

Instruction Execution Detail



Cy	IM ≤ Rd			IM > Rd		
	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011	prevA	(prevA)	1011
2–8	prevA	(prevA)	1011	prevA	(prevA)	1011
9	OpA + 4	IEW	1000	prevA	(prevA)	1011
10				OpA + 4	IEW	1000

Syntax **JMP**

Execution

JMP *Rd*	destination → (PC)
JMP *add*	destination → (PC)
JMP **disp[Rd]*	disp + (Rd) → (PC)
JMP **Rd*	disp + (Rd) → (PC) with *disp* = 0000h

Modes Supported

Rd
addr
**disp16[Rd]*
**Rd* (assembles as JMP **0000h[Rd]*)

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description

Jump to the destination operand. *(For jump to a **subroutine**, see the CALL instruction, page 5-34.)*

JMP ***Rd*** jumps to the *word address* value (see note below) contained in register Rd (i.e., *Rd* contents → (PC)).

JMP ***addr*** jumps to the 17-bit *address* location (one half its value stored in the extension word as a 16-bit *word address*).

JMP ***disp16[Rd]*** and **JMP** ***Rd** (the latter is assembled as if written JMP **0h[Rd]*) use the following steps to derive the destination:

1) For **disp16[Rd]*, add the displacement (*disp*) and the contents of *Rd* to compute a memory address (displacement value can be 0–FFFFh).

2) At this memory address, obtain a *word* address and apply this to the PC. In turn, this value is applied to the address bus as a 17-bit address. Note that this word address must be half the destination address-bus value. A graphic explanation of this instruction is shown in Figure 4–12 on page 4-15.

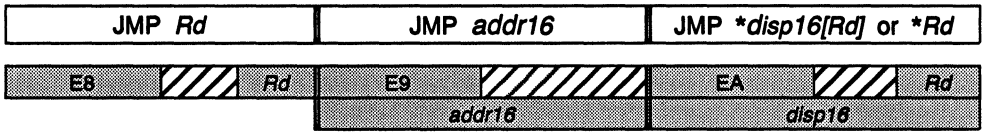
Note: 16-Bit *Word Address* Translates to 17-Bit Address Bus

The *word address* is a 16-bit value transformed to a 17-bit memory address, via the program counter, by overlaying data bits 0–15 onto address lines A16–A1 and forcing A0 to 0. This is further explained in Section 2.2 and its associated figures (page 2-4). Figure 4–13 (page 4-16) shows how to set the word address using the ? operator.

Examples

```
LABEL   JMP    *R8          ; Jump to the address of
                            ;  ((R8)) * 2.
        jmp    &code7       ; Jump to the address of
                            ;  code7.
        JMP    *extra[R7]   ; Jump to the address of
                            ;  (extra + (R7)) * 2
```

Instruction Execution Detail



Cy	Address	Data	w̄ b f̄ d̄	Address	Data	w̄ b f̄ d̄	Address	Data	w̄ b f̄ d̄
1	prevA	(prevA)	1011	prevA	(prevA)	1011	prevA	(prevA)	1011
2	2Rd	IW	1001	2caddr	IW	1001	disp + Rd	caddr	1011
3	(2Rd) + 2	IEW	1000	(2caddr) + 2	IEW	1000	2caddr	IW	1001
4							(2caddr) + 2	IEW	1000

Syntax **LDBIT{B}**

Execution value of bit in destination (specified by source mask) → C[[ST]]

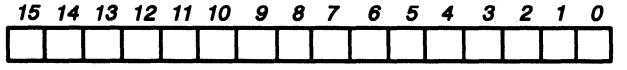
Modes Supported

*#imm4,*disp16[Rd]*	*(byte only)*
*Rs,*disp16[Rd]*	*(byte only)*
Rs,Rd	*(word only)*
#imm4,Rd	*(word only)*

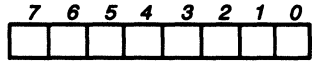
Status Bits

Z unchanged
N unchanged
C equals value of loaded bit
V unchanged

Description The value of a bit in the destination, specified by the source operand, is placed in the carry bit of the status register. The **source operand** is a value in the range 0–15 contained either in the four least significant bits of a register or as an immediate value. Bit numbers correspond to the following formats:



= word bit numbering



= byte bit numbering

When the destination is a memory address, *only byte memory accesses* are performed. Useful values for **byte operations** are 0–7, which select one of the eight bits. When a value in the range 8–15 is used in byte operations, the addressed byte is read, but the C[[ST]] bit is left equal to 0. Useful values for **word operations** are 0–15.

LDBIT is intended to be used with a BNC (*branch if carry clear* — C[[ST]] equals 0) or a BC (*branch if carry set* — C[[ST]] equals 1) instruction.

Example Check if the most significant bit of byte (or word) address A000h is a 1:

```
LDBITB   #15,*A000h[ZR]   ; Place MSB of addr A000h
                          ;  in C bit of status reg.
BC       A_ONE            ; Branch if MSB = 1
                          ;  (C bit = 1)
```

Instruction Execution Detail

LDBIT # *imm4,Rd*	LDBIT *Rs,Rd*	LDBITB *#imm4,*disp16[Rd]*
94 (w) | *imm4* | *Rd*	E4 (w) | *Rs* | *Rd*	95 (b) | *imm4* | Rd *disp16*

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d	Address	Data	$\overline{w}$ b $\overline{f}$ d	Address	Data	$\overline{w}$ b $\overline{f}$ d
1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1	OpA + 4	IW	1 0 0 1
2	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1	disp + Rd	(disp + Rd)	1 1 1 1
3				OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
4							OpA + 6	IEW	1 0 0 0

LDBITB *Rs,*disp16[Rd]*
E5 (b) | *Rs* | Rd *disp16*

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d
1	prevA	(prevA)	1 0 1 1
2	OpA + 4	IW	1 0 0 1
3	disp + Rd	(disp + Rd)	1 1 1 1
4	prevA	(prevA)	1 0 1 1
5	OpA + 6	IEW	1 0 0 0

Syntax **LDEA**

Execution displacement value + (*Rs*) → (*Rd*)

Mode Supported **disp16[Rs],Rd*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description Load the destination register with the sum of the source-register contents plus the 16-bit displacement (offset) value. Note the following:

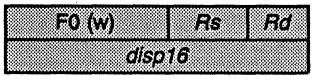
- If *disp16* is a label, its value is the memory address value of *disp16* (not the contents of *disp16*).
- If *Rs* is the ZR (zero register), then execution loads *only* the value of *disp16* into *Rd*.

Examples

```
LABEL1  LDEA   *BUFF7[r2],R8 ; Load value of BUFF7 plus
                             ;  the contents of R2 into
                             ;  register R8.
LABEL2  LDEA   *BUFF7[zr],R8 ; Load value of BUFF7 into
                             ;  register R8.
LABEL3  LDEA   &BUFF7,R8     ; Load value of BUFF7 into
                             ;  register R8 (assembles
                             ;  to same code as shown
                             ;  for LABEL2 instruction).
```

Instruction Execution Detail

LDEA **disp16[Rs],Rd*



Cy	Address	Data	$\overline{w}$ b $\overline{i}$ $\overline{d}$
1	OpA + 4	IW	1001
2	OpA + 6	IEW	1000

Syntax **LIMHS{B}**

Execution

IF [{V[[ST]] = 1 *and* N[[ST]] = 1} **or** {V[[ST]] = 0 *and* (source) < (Rd)}] ,
THEN
(source) → (Rd)
zero → V[[ST]]
one → C[[ST]]
ELSE no change to Rd

Mode Supported **disp16[Rs],Rd*

Status Bits

Z unchanged if Rd is not modified; otherwise, set if the contents of Rd are zero, and cleared if the contents of Rd are nonzero
N equals the most significant bit of Rd if Rd is modified; otherwise, unchanged
C set if Rd is modified; otherwise, unchanged
V cleared if Rd is modified; otherwise, unchanged

Description

This instruction ensures that a register variable remains less than or equal to its maximum legal value.

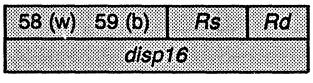
This instruction leaves the destination register with either its original contents or the value given by the contents of the source operand. The C[[ST]] bit declares that the contents of Rd has been modified. Two conditions warrant setting the register to the contents of the source operand:

- Upon entry, V[[ST]] = 1 and N[[ST]] = 1, which indicates that an unsigned overflow occurred before this instruction.
- The signed data value at the source operand is less than the signed contents of Rd.

Byte operations test only the least significant byte of a register. If a byte in Rd is modified, the most significant byte of Rd is cleared. Status bits are set with respect to the size (byte or word) of the operation.

Instruction Execution Detail

LIMHS{B} **disp16[Rs],Rd*



	V[[ST]] = 1			V[[ST]] = 0		
Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2	disp + Rs	(disp + Rs)	1 S 1 1	disp + Rs	(disp + Rs)	1 S 1 1
3	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
4	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
5	OpA + 6	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
6				OpA + 6	IEW	1 0 0 0

Syntax **LIMHU{B}**

Execution

IF [C[[ST]] = 1 **or** (source) < (Rd)] ,
  THEN
    source → (Rd)
    one → V[[ST]]
ENDIF
zero → C[[ST]]
IF a *byte* instruction (LIMHUB),
  THEN
    zeroes → Rd bits 8–15
ENDIF
ELSE (Rd) remains unchanged

Mode Supported **disp16[Rs],Rd*

Status Bits

Z set if the contents of Rd are zero; otherwise, cleared
N equals the most significant bit of Rd
C cleared
V set if Rd is modified; otherwise, unchanged

Description Use this instruction to ensure that a register variable remains less than or equal to its maximum legal value.

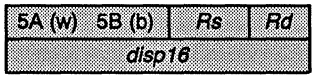
This instruction leaves a register with either its original contents or the value given by the contents of the source operand. The V[[ST]] bit declares that the contents of Rd have been modified. Two conditions warrant setting the register to the contents of the source operand:

- ❑ Upon entry, C[[ST]] = 1, indicating that an unsigned overflow occurred before this instruction.
- ❑ The unsigned data value at the source operand is less than the unsigned contents of Rd.

Byte operations test only the least significant byte of a register and *always clear* the most significant byte of Rd. Status bits are set with respect to the size (byte or word) of the operation.

Instruction Execution Detail

LIMHU{B} **disp16[Rs],Rd*



	C[[ST]] = 1			C[[ST]] = 0		
Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2	disp + Rs	(disp + Rs)	1 S 1 1	disp + Rs	(disp + Rs)	1 S 1 1
3	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
4	OpA + 6	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
5				OpA + 6	IEW	1 0 0 0

Syntax **LIMLS{B}**

Execution

IF [{V[[ST]] = 1 *and* N[[ST]] = 0} **or** {V[[ST]] = 0 *and* (source) > (Rd)}],
THEN
(source) → (Rd)
zero → V[[ST]]
one → C[[ST]]
ELSE (Rd) remains unchanged

Mode Supported **disp16[Rs],Rd*

Status Bits

Z unchanged if Rd is not modified; otherwise, *set* if the contents of Rd are zero, and *cleared* if the contents of Rd are nonzero
N equals the most significant bit Rd if Rd is modified; otherwise, unchanged
C set if Rd is modified; otherwise, unchanged
V cleared if Rd is modified; otherwise, unchanged

Description

This instruction leaves the destination register with either its original contents or the value given by the contents of the source operand. The C[[ST]] bit declares that the contents of Rd has been modified. Either of two conditions set the register to the contents of the source operand:

- Upon entry, V[[ST]] = 1 and N[[ST]] = 0, indicating that an unsigned overflow occurred before this instruction.
- Or when V[[ST]] = 0, *and* the signed data value at the source operand is greater than the signed contents of Rd.

Use this instruction to ensure that a register variable remains greater than or equal to its minimum legal value.

Byte operations test only the least significant byte of a register. If a byte in Rd is modified, the most significant byte of Rd is cleared. Status bits are set with respect to the size (byte or word) of the operation.

Instruction Execution Detail

LIMLS{B} **disp16[Rs],Rd*

5C (w) 5D (b)	*Rs*	*Rd*
disp16		

	V[[ST]] = 1			V[[ST]] = 0		
Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2	disp + Rs	(disp + Rs)	1 S 1 1	disp + Rs	(disp + Rs)	1 S 1 1
3	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
4	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
5	OpA + 6	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
6				OpA + 6	IEW	1 0 0 0

Syntax **LIMLU{B}**

Execution

IF [C[[ST]]= 1 **or** (source) > (Rd)],
THEN
(source) → (Rd)
1 (one) → V[[ST]]
ENDIF
0 → C[[ST]]

IF a byte instruction (LIMLUB),
THEN 0 → (Rd8-Rd15)
ENDIF

Mode Supported **disp16[Rs],Rd*

Status Bits

Z set if the contents of Rd are zero; otherwise, cleared
N equals the most significant bit of Rd
C cleared
V set if Rd is modified; otherwise, unchanged

Description This instruction will leave the destination register with either its original contents or the value given by the contents of the source operand. The V[[ST]] bit declares that the contents of Rd have been modified. Two conditions warrant setting the register to the contents of the source operand:

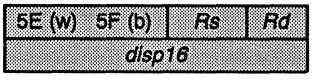
- ❑ Upon entry, C[[ST]] = 1, which indicates that an unsigned overflow occurred prior to this instruction.
- ❑ The unsigned data value at the source operand is greater than the unsigned contents of Rd.

Use this instruction to ensure that a register variable remains greater than or equal to its minimum legal value.

Byte operations test only the least significant byte of a register. If a byte in Rd is modified, the most significant byte of Rd is cleared. Status bits are set with respect to the size (byte or word) of the operation.

Instruction Execution Detail

LIMLU{B} **disp16[Rs],Rd*



	C[[ST]] = 1			C[[ST]] = 0		
Cy	Address	Data	$\overline{w}$ b f $\overline{d}$	Address	Data	$\overline{w}$ b f $\overline{d}$
1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2	disp + Rs	(disp + Rs)	1 S 1 1	disp + Rs	(disp + Rs)	1 S 1 1
3	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
4	OpA + 6	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
5				OpA + 6	IEW	1 0 0 0

Syntax **LINK**

Execution

(FP) → ((SP))
(SP) → (FP)
(SP) + 2 → (SP)
(SP) + (2 × displacement) → (SP)

Mode Supported *disp8*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

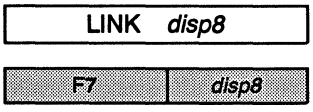
Description This instruction links the frame pointer (FP) to the current system stack frame by executing these steps:

1) Push the FP contents onto the system stack.
2) Set the FP to the SP value.
3) Allocate a *displacement* amount of words on the stack.

The 8-bit, unsigned, immediate displacement value is multiplied by 2 before being added to the SP, in order to keep the value of SP even.

A stack frame of 0 to 255 *words* can be allocated.

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	SP	FP	0 0 1 1
2	prevA	(prevA)	1 0 1 1
3	prevA	(prevA)	1 0 1 1
4	OpA + 4	IEW	1 0 0 0

Syntax **LSR{L}**

Execution Shift destination register to the right by a source-count amount of bits

Modes Supported

Rs,Rd	*(word)*
#imm4,Rd	*(word)*
Rs,IM:Rd	*(longword)*
#imm4,IM:Rd	*(longword)*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C equals the last bit shifted out of the register; cleared if the shift count in Rs is zero
V cleared

Description Logically right-shift the destination register's contents by the number of the bit count in the source operand. At the same time, shift (the same number of) zero(es) into the most significant bit(s). A four-bit field in the source operand contains the shift count of 0–15.

For shifts using **immediate values**, *source-operand immediate values* of 0001_2 to 1111_2 correspond to shift counts of 1–15, with a field value of 0000_2 representing a shift of 16. For shift values **in Rs**, the four LSBs of 0000_2 to 1111_2 represent a shift of 0 to 15.

Status bits are set with respect to the size (word or longword) of the operation. Longword shifts imply the use of register IM as the most significant word of the 32-bit object. The result of LSRL s,IM:IM is undefined.

Examples

```
LABEL    MOV #6,R12        ; Place value of 6 in R12
         LSR R12,R3        ; Shift R3 to the right 6 bits
                           ;   and fill the 6 MSBs with
                           ;   zeroes.
                           ;
SHIFT_2  LSR #2,R8         ; Logically shift R8 by 2 bits.
                           ;
SHIFT_12 LSRL #12,IM:R5    ; Logically right-shift the
                           ;   concatenated values in IM:R5
                           ;   by 12.
                           ;
         LSRL R3,IM:R2     ; The count value of the 4 LSBs
                           ;   of R3 are used to logically
                           ;   right-shift the IM:R2
                           ;   register pair.
```

Instruction Execution Detail

LSR *#imm4,Rd*

BC (w)	*imm4*	*Rd*

Cycle/ Period†	Address	Data	$\overline{w}$ b $\overline{i}$ $\overline{d}$
n (repeat)	prevA	(prevA)	1 0 1 1
n + 1	OpA+4	IEW	1 0 0 0

LSR *Rs,Rd*

BE (w)	*Rs*	*Rd*

Cycle/ Period†	Address	Data	$\overline{w}$ b $\overline{i}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2	prevA	(prevA)	1 0 1 1
n (repeat)	prevA	(prevA)	1 0 1 1
n+1	OpA+4	IEW	1 0 0 0
	Total cycles: *n* + 3		

LSRL *#imm4,IM:Rd*

BD (L)	*imm4*	*Rd*

Cycle/ Period†	Address	Data	$\overline{w}$ b $\overline{i}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2*n*–2 (repeat)	prevA	(prevA)	1 0 1 1
2*n*	OpA+4	IEW	1 0 0 0

LSRL *Rs,IM:Rd*

BF (L)	*Rs*	*Rd*

Cycle/ Period†	Address	Data	$\overline{w}$ b $\overline{i}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2	prevA	(prevA)	1 0 1 1
3	prevA	(prevA)	1 0 1 1
2*n*–2 (repeat)	prevA	(prevA)	1 0 1 1
2*n*	OpA+4	IEW	1 0 0 0
	Total cycles: 2 + 2*n*, or 3 if *Rs* = xxx0h		

† A single number represents a *given cycle*; an expression of *n* represents a *cycle* or *period of cycles*, depending on the n^{th} number of shifts or repeats.

Syntax **MOV{B}**

Execution (source) → (destination)

Modes Supported

Rs,Rd
*Rs,*Rd*
*Rs,*Rd+*
*Rs,*disp16[Rd]*

**Rs,Rd*
**Rs,*Rd*
**Rs,*Rd+*
**Rs,*disp16[Rd]*

**Rs+,Rd*
**Rs+,*Rd*
**Rs+,*Rd+*
**Rs+,*disp16[Rd]*

#imm16,Rd
*#imm16,*Rd*
*#imm16,*Rd+*
*#imm16,*disp16[Rd]*

**disp16[Rs],Rd*
**disp16[Rs],*Rd*
**disp16[Rs],*Rd+*
**disp_s16[Rs],*disp_d16[Rd]*

**–Rs,Rd*

Status Bits

Z set if the transferred data was zero; otherwise, cleared
N equals the most significant bit of the transferred data
C unchanged
V cleared

Description Transfer data from the source operand to the destination operand.

When byte data is moved to a register, the least significant byte receives the data, while the most significant byte is cleared. When data is moved from a register, only the least significant byte of the register is moved.

Status bits are set with respect to the size (byte or word) of the operation.

Note: Use FMOV to Address 0 – 1FFFFh (Up to 128K Bytes)

The MOV instruction moves (copies) between registers or from/to an address space within the first 64K bytes. Use the FMOV instruction to move data in the address space from 0 to 128K bytes.

Four special cases exist when the source operand is *(Rn)+ and the destination operand is a mode using *the same register*.

Instruction		Operation (see Note)
MOV	**Rn+,Rn*	((Rn)) + size → (Rn)
MOV	**Rn+,*Rn*	((Rn)) → ((Rn)); (Rn) + size → (Rn)
MOV	**Rn+,*Rn+*	(Rn) → (Rn + size); Rn + 2 × size → Rn
MOV	**Rn+,*d[Rn]*	(Rn) → (d + (Rn)); Rn + size → Rn

Note: The "size" is the increment size (1 for byte, 2 for word).

Examples

```
Label   mov     r2,r3       ; Move contents of R2 to R3.
                            ;
clr_R5  mov     zr,r5       ; Clear contents of R5
                            ;
        mov     zr,*r9+     ; Clear the contents at the
                            ;   address in R9 then incre-
                            ;   ment R9.
                            ;
        MOV *50h[r1],r2     ; Compute the source address
                            ;   by adding 50h and the contents
                            ;   of R1. Move the contents at
                            ;   this address to R2.
                            ;
        movb #1234h,*r10    ; Move the value 34h into the
                            ;   indirect contents of r10.
                            ;   Force MS byte of r10 to
                            ;   all zeroes.
                            ;
        mov &BUF1,&BUF2     ; Move the word at location BUF1
                            ;   to word location BUF2.
                            ;
        MOVB &LOC1,&LOC2    ; Move the byte value at LOC1
                            ;   to byte address LOC2.
                            ;
        movb *-r7,im        ; Decrement value in R7; the
                            ;   result is the source add-
                            ;   ress. Move the LS byte at
                            ;   this address to the IM with
                            ;   the MS byte of the IM = 0.
```

Instruction Execution Detail

MOV{B} *Rs, Rd*	MOV{B} *Rs, *Rd*	MOV{B} *Rs,*Rd+*
02 (w) 03 (b) | *Rs* | *Rd*	04 (w) 05 (b) | *Rs* | *Rd*	06 (w) 07 (b) | *Rs* | *Rd*

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	Rd	Rs	0 S 1 1	Rd	Rs	0 S 1 1
2				OpA + 4	IEW	1 0 0 0	OpA + 4	IEW	1 0 0 0

MOV{B} *Rs,*disp16[Rd]*	MOV{B} **Rs,Rd*	MOV{B} **Rs,*Rd*
08 (w) 09 (b) | *Rs* | *Rd* *disp16*	0B (w) 0A (b) | *Rs* | *Rd*	0C (w) 0D (b) | *Rs* | *Rd*

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IW	1 0 0 1	RS	(Rs)	1 S 1 1	Rs	(Rs)	1 S 1 1
2	disp + Rd	Rs	1 0 1 1	OpA + 4	IEW	1 0 0 0	Rd	(Rs)	0 S 1 1
3	OpA + 6	IEW	1 0 0 0				OpA + 4	IEW	1 0 0 0

MOV{B} **Rs,*Rd+*	MOV{B} **Rs,*disp16[Rd]*	MOV{B} **Rs+,Rd*
0E (w) 0F (b) | *Rs* | *Rd*	10 (w) 11 (b) | *Rs* | *Rd* *disp16*	12 (w) 13 (b) | Rs | Rd

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	Rs	(Rs)	1 S 1 1	OpA + 4	IW	1 0 0 1	Rs	(Rs)	1 S 1 1
2	Rd	(Rs)	0 S 1 1	Rs	(Rs)	1 S 1 1	prev	(prev)	1 0 1 1
3	OpA + 4	IEW	1 0 0 0	disp + Rd	(Rs)	0 S 1 1	OpA + 4	IEW	1 0 0 0
4				OpA + 6	IEW	1 0 0 0			

MOV{B} **Rs+,*Rd*	MOV{B} **Rs+,*Rd+*	MOV{B} **Rs+,*disp16[Rd]*
14 (w) 15 (b) | *Rs* | *Rd*	16 (w) 17 (b) | *Rs* | Rd	18 (w) 19 (b) | *Rs* | Rd *disp16*

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	Rs	(Rs)	1 S 1 1	Rs	(Rs)	1 S 1 1	Rs	(Rs)	1 S 1 1
2	Rd	(Rs)	0 S 1 1	prev	(prev)	1 0 1 1	OpA + 4	IW	1 0 0 1
3	OpA + 4	IEW	1 0 0 0	Rd	(Rs)	0 S 1 1	disp + Rd	(Rs)	0 S 1 1
4				OpA + 4	IEW	1 0 0 0	OpA + 6	IEW	1 0 0 0

MOV{B} *#imm16,Rd*	MOV{B} *#imm16,*Rd*	MOV{B} *#imm16,*Rd+*
1A (w) 1B (b) / *Rd*	1C (w) 1D (b) / *Rd*	1E (w) 1F (b) / *Rd*
imm16	*imm16*	*imm16*

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IW	1 0 0 1	Rd	data	0 S 1 1	Rd	data	0 S 1 1
2	OpA + 6	IEW	1 0 0 0	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
3				OpA + 6	IEW	1 0 0 0	OpA + 6	IEW	1 0 0 0

MOV{B} *#imm16,*disp16[Rd]*	MOV{B} **disp16[Rs],Rd*	MOV{B} **disp16[Rs],*Rd*
20 (w) 21 (b) / *Rd*	22 (w) 23 (b) / *Rs* / *Rd*	24 (w) 25 (b) / *Rs* / *Rd*
disp16	*disp16*	*disp16*
imm16		

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	data	1 0 0 1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2	disp + Rd	data	0 S 1 1	disp + Rs	(disp + Rs)	1 S 1 1	disp + Rd	(disp + Rs)	1 S 1 1
3	OpA + 6	IW	1 0 0 1	OpA + 6	IEW	1 0 0 0	Rd	(disp + Rs)	0 S 1 1
4	OpA + 8	IEW	1 0 0 0				OpA + 6	IEW	1 0 0 0

MOV{B} **disp16[Rs],*Rd+*	MOV{B} **disp_s16[Rs],*disp_d16[Rd]*	MOV{B} **–Rs,Rd*
26 (w) 27 (b) / *Rs* / *Rd*	28 (w) 29 (b) / *Rs* / *Rd*	2A (w) 2B (b) / *Rs* / *Rd*
disp16	*disp116*	
	disp216	

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IW	1 0 0 1	OpA + 4	disp2	1 0 0 1	prev	(prev)	1 0 1 1
2	disp + Rs	(disp + Rs)	1 S 1 1	OpA + 6	IW	1 0 0 1	Rs - S	(Rs - S)	1 S 1 1
3	Rd	(disp + Rs)	0 S 1 1	disp1 + Rs	(disp1 + Rs)	1 S 1 1	OpA + 4	IEW	1 0 0 0
4	OpA + 6	IEW	1 0 0 0	disp2 + Rd	(disp1 + Rd)	0 S 1 1			
5				OpA + 8	IEW	1 0 0 0			

Syntax **MOVQ**

Execution immediate data value → (destination)

Mode Supported *#imm4,Rd* *(4-bit value entered, zero extended; 16-bit word moved)*

Status Bits
Z set if the transferred data was zero; cleared otherwise
N cleared
C unchanged
V cleared

Description Transfer quick immediate data to the destination operand. Quick immediate data is a *4-bit* value of 0–15, that has been *zero-extended to word.* This instruction requires **one** word and operates in one cycle; whereas, MOV *#data,Rd* takes up **two** words and two cycles.

Note that a MOVQB is unnecessary, because MOVQ generates the same result in destination register *Rd* with the register's MSbyte cleared to zeroes.

Example
```
LABEL   MOVQ  #3,R12     ; Load the value 3 into R12.
```

Instruction Execution Detail

MOVQ *#imm4 ,Rd*

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d
1	OpA + 4	IEW	1 0 0 0

Syntax **MPYBWU**

Execution (LSbyte of Rs x Rd) + 80h → (Temp) (8 x 16 → 24 + 000080h)
(Temp) ÷ 256 → (Rd)

Mode Supported *Rs,Rd*

Status Bits
Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C cleared
V cleared

Description Multiply the 8-bit value in the least significant byte of Rs *by* the 16-bit value in Rd. Add 000080h to the 24-bit intermediate product, and place the most significant word of the sum in Rd.

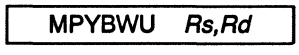
During the multiply, the most significant byte of Rs is ignored, and the contents of Rs are left unchanged.

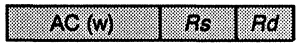
The internal multiply is 8 bits x 16 bits, which generates a 24-bit intermediate result. Typically, this instruction is used when the 8-bit value in Rs is a fraction and Rd holds an integer. Hence, the product is a fixed point value with the integer portion in bits 8–23 and the fraction in bits 0–7. The value 000080h is added to this temporary product to round it back to an integer, yielding a rounded integer value in bits 8–23. This rounded result is then placed in Rd.

Examples

```
label   MPYBWU R7,R8     ; Multiply the LS byte of R7
                         ;  by R8, then add 80h to the
                         ;  product. Place the MS word
                         ;  of this result in R8.
```

Instruction Execution Detail





Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d
1–6	prevA	(prevA)	1011
7	OpA + 4	IEW	1000

Syntax **MPYS{B}**

Execution Rs value × (destination) → (destination)

Modes Supported

Rs,Rd	[*byte only (8 × 8 → 16):*	*Rs × Rd → Rd*]
Rs,IM:Rd	[*word only (16 × 16 → 32):*	*Rs × Rd → IM:Rd*]

Status Bits

Z set if the product is zero; cleared otherwise
N equals the most significant bit of the product
C cleared
V cleared

Description Perform a multiply of the signed contents of the destination register by the signed contents of the source register. The product of byte multiplication is placed in the destination register. The most significant word of the product of word multiplication is placed in the *IM* register, and the least significant word is placed in the destination register.

The result assignment sequence places the most significant word of the product in the *IM* first and the least significant word to *Rd* second. If *Rd* is *also IM* (for example, **MPYS** *Rs,IM:IM*), then the most significant word in *IM* is overwritten by the least significant word.

Signed overflow on a multiple occurs when the product cannot be successfully truncated to the size of the operands without data loss. For **MPYSB**, this occurs when bits 15–7 of the product are not equal, and for **MPYS** when bits 31–15 of the product are not equal. To detect this condition, follow an MPYS instruction with a TRUNCS instruction. This sequence will leave V[[ST]] and N[[ST]] correctly set for such signed overflows.

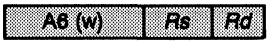
Status bits are set with respect to the size (word or longword) of the product.

Examples

```
LABEL   MPYSB  R3,R4   ; Multiply (signed) the LS bytes
                       ;  of R3 and R4. Store result in
                       ;  R4.
MULT  MPYS R8,IM:R9    ; Multiply (signed) R8 by R9.
                       ;  Place result in the concat–
                       ;  tenated IM:R9 register pair.
```

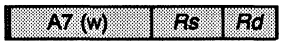
Instruction Execution Detail

MPYS *Rs,IM:Rd*



Cy	Rd ≥ 0			Rd < 0		
	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1–12	prevA	(prevA)	1011	prevA	(prevA)	1011
13	OpA + 4	IEW	1000	prevA	(prevA)	1011
14				OpA + 4	IEW	1000

MPYSB *Rs,Rd*



Cy	Rd ≥ 0			Rd < 0		
	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011	prevA	(prevA)	1011
2	prevA	$Rd_{LSB} \times 256$	0011	prevA	$Rd_{LSB} \times 256$	0011
3–8	prevA	(prevA)	1011	prevA	(prevA)	1011
10	OpA + 4	IEW	1000	prevA	$Rd_{LSB} \times 256$	1011
11				OpA + 4	IEW	1000

Syntax **MPYU{B}**

Execution Rs value × (destination) → (destination)

Modes Supported

Rs,Rd	[*byte only (8 × 8 → 16):*	*Rs × Rd → Rd*]
Rs,IM:Rd	[*word only (16 × 16 → 32):*	*Rs × Rd → IM:Rd*]

Status Bits

Z set if the product is zero; cleared otherwise
N equals the most significant bit of the product
C cleared
V cleared

Description Perform an unsigned multiply of the unsigned contents of the destination register by the unsigned contents of the source register. The product of *byte* multiplication is placed in the destination register. The most significant word of the product of *word* multiplication is placed in the *IM* register, and the least significant word in the destination register.

The result assignment sequence places the most significant word of the product in register *IM* first and then the least significant word to *Rd* second. If *Rd* is also the *IM* (for example, **MPYU** *Rs,IM:IM*), then the most significant word in *IM* is overwritten by the least significant word.

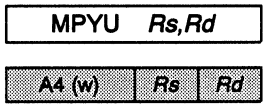
Unsigned overflow on a multiply occurs when the product cannot be successfully truncated to the size of its operands without data loss. For **MPYUB,** this occurs when bits 15–8 of the product are not zero and for **MPYU** when bits 31–16 of the product are not zero. To detect this condition, follow an MPYUB instruction with a TRUNCU instruction, or follow an MPYU instruction with a CMP *IM,ZR*. These sequences will leave C[[ST]]=1 for the signed overflows.

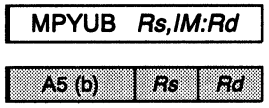
Status bits are set with respect to the size (word or longword) of the product.

Examples

```
label   mpyub r3,r4      ; Multiply (unsigned) the LS
                         ;  bytes of R3 and R4. Store
                         ;  the result in R4.
                         ;
mult    mpyu r8,IM:r9    ; Multiply (unsigned) R8 by R9.
                         ;  Store results in the IM:r9
                         ;  concatenated register pair.
```

Instruction Execution Detail





Cy	Address	Data	w b f d
1–12	prevA	(prevA)	1 0 1 1
13	OpA + 4	IEW	1 0 0 0

Cy	Address	Data	w b f d
1	prevA	(prevA)	1 0 1 1
2	prevA	$Rd_{LSB} \times 256$	0 0 1 1
3–8	prevA	(prevA)	1 0 1 1
8	OpA + 4	IEW	1 0 0 0

Syntax **NOP** *Synthetic Instruction: Executes as SBIT0 #15,ZR*

Execution zero → bit 15 of ZR
(same as SBIT0 *#15,ZR*)

Mode Supported *Operand not necessary for NOP*

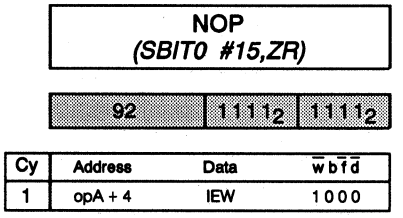
Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description Perform no operation; CPU state is unchanged except for advancement of the PC to the next instruction address. This instruction takes one cycle.

Example

```
DELAY   NOP          ; Causes one cycle delay
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b f $\overline{d}$
1	opA + 4	IEW	1000

Syntax **NOT{B}** *Synthetic Instruction: Executes as XNOR{B} ZR,Rd*

Execution NOT((source) XOR (destination)) → (destination)

(executes same as XNOR{B} *ZR,Rd*)

Mode Supported *Rd*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C unchanged
V cleared

Description Perform a ones complement on the destination register's contents. Effectively, XOR (exclusively OR) the Rd with the all-zero ZR, then take a ones complement of the result (XNOR the Rd).

For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will always be FFh.

Status bits are set with respect to the size (byte or word) of the operation.

Examples

```
label   NOTB   IM        ; Invert LS byte of the IM
                         ;
Invert  NOT    R12       ; Invert R12
```

Instruction Execution Detail

NOT{B} *Rd*
(XNOR{B} ZR,Rd)

2C (w) 2D (b)	1111_2	*Rd*

Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	OpA + 4	IEW	1 0 0 0

Syntax **OR{B}**

Execution (source) OR (destination) → (destination)

Modes Supported
Rs,Rd
*Rs,*disp16[Rd]*
#imm16,Rd
*#imm16,*disp16[Rd]*

Status Bits
Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C unchanged
V cleared

Description Logically inclusive OR the contents of the source operand with the contents of the destination operand.

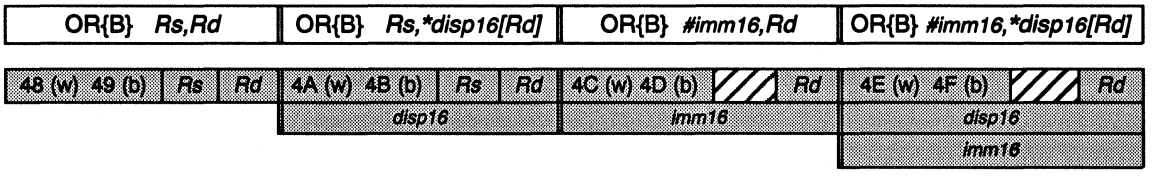
For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will always be 00h. Nonregister destinations receive the least significant byte of the result, while registers receive the entire word.

Status bits are set with respect to the size (byte or word) of the operation.

Examples

```
Label   OR      R5,R6       ; Logically OR the contents of
                            ;  R5 and R6. Store the value
                            ;  in R6.
                            ;
Set_2   ORB     #4h,&FLAG   ; Set bit 2 of location FLAG.
                            ;
Set_8   OR #EIGHT,&Flag     ; OR mask value EIGHT with
                            ;  location Flag.
```

Instruction Execution Detail



Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1	OpA + 4	data	1 0 0 1
2				disp + Rd	(disp + Rd)	1 S 1 1	OpA + 6	IEW	1 0 0 0	disp + Rd	(disp + Rd)	1 S 1 1
3				prevA	(prevA)	1 0 1 1				OpA + 6	IW	1 0 0 1
4				disp + Rd	result	0 S 1 1				disp + Rd	result	0 S 1 1
5				OpA + 6	IEW	1 0 0 0				OpA + 8	IEW	1 0 0 0

Syntax **POP**

Execution

FOR INDEX = Register_Last to Register_First BY –1,
DO (SP) – 2 → (SP)
((SP)) → (register$_{(index)}$)

Mode Supported *RL,RF* *(Register_Last and Register_First; Required: RL ≥ RF)*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description

Starting at the SP value, pull words from the system stack into the register(s) starting with *RL* (Register_Last) through and including *RF* (Register_First). The enumerator of *RL* must be arithmetically greater than or equal to the enumerator of *RF* (for example, POP R7,R5 is valid because 7 ≥ 5).

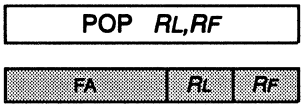
This instruction is most efficiently used for minimizing code space and execution time when restoring CPU context. It is faster than multiple MOV *–SP,Rd instructions and consumes only two bytes of program memory.

If the SP (R13) is included in the *RL–RF* range, the value retrieved and placed into the SP will take effect immediately and will direct the retrievals of any remaining registers in the range. Care must be taken to ensure that the SP value on the system stack will not corrupt the current context. A matched pair of instructions, PUSH RF,RL and POP RL,RF, executes correctly when the SP is included within the range.

Example

```
Label    POP    R12,R2   ; Pop the contents of the Stack
                         ;  into registers R12 to R2
```

Instruction Execution Detail



Cy†	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
n_1	prevA	(prevA)	1 0 1 1
n_2	SP - 2n	(SP - 2n)	1 0 1 1
2n + 1	OpA + 4	IEW	1 0 0 0

† Both n_1 and n_2 are repeated during the exchange of one stack register to memory. The total number of exchanges (stack words stored) is multiplied by 2 and incremented for the cycle total (last line).

Syntax **PUSH**

Execution

FOR *index* = Register_First to Register_Last BY +1,
DO (register$_{(index)}$) → ((SP))
(SP) + 2 → (SP)

Mode Supported *RF,RL* *(Register_First and Register_Last; Required: $R_L \geq R_F$)*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

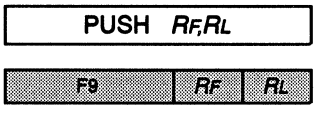
Description Push register contents onto the system, stack starting at the SP value. Registers to push are from *RF* (register_first) through and including *RL* (register_last). The enumerator of *RL* must be arithmetically greater than or equal to the enumerator of *RF* (for example, PUSH *R5,R7* is valid because $7 \geq 5$).

This instruction is most efficiently used for minimizing code space and execution time when saving CPU contexts. It is always faster than multiple `MOV Rs,*SP+` instructions and consumes only two bytes of program memory.

Example

```
SAVE    PUSH    R6,R3    ; Push the contents of R6, R5,
                         ;  R4, and R3 onto the Stack
```

Instruction Execution Detail



Cy†	Address	Data	w b f d
n (repeat)	SP + 2(n-1)	(reg)	0011
n + 1	OpA + 4	IEW	1000

† The n is the number of register-to-stack exchanges.

Syntax **RTDU**

Execution

(FP) – 2 → (SP)
((FP)) → (FP)
((SP)) → (PC)
(SP) – 2displacement → (SP)

Mode Supported *disp8*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description

Unlink and deallocate the current system stack frame:

1) Load SP with the contents of the frame pointer (FP),
2) Retrieve the previous value of FP from the system stack,
3) Pull the return address from the system stack and place it in the PC, and
4) Deallocate additional stack space by subtracting the 8-bit unsigned word displacement from the value of SP.

Note that since the 8-bit value is a word displacement, it is internally multiplied by two to generate an even value and to keep the SP word aligned.

The return address is a *word address* that is transformed to a 17-bit physical memory address, via the program counter, by overlaying data bits 15–0 onto address lines A16–A1 and forcing A0 to 0. This instruction can be a return mechanism for a CALL subroutine *if and only if* the subroutine executed a LINK instruction and did not execute an UNLINK instruction.

Instruction Execution Detail

RTDU *disp8*	
F8	*disp8*

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2	FP	old FP	1 0 1 1
3	FP - 2	Rtn A + 2	1 0 1 1
4	Rtn A	IW	1 0 0 1
5	Rtn A + 2	IEW	1 0 0 0

Syntax **RTI**

Execution

(SP) – 2 → (SP)
((SP)) → (PC)
(PC) – 2 → (PC)
(SP) – 2 → (SP)
((SP)) → (ST)

Modes Supported *Operand not necessary for RTI*

Status Bits

Z reflects the status data pulled from the system stack
N reflects the status data pulled from the system stack
C reflects the status data pulled from the system stack
V reflects the status data pulled from the system stack

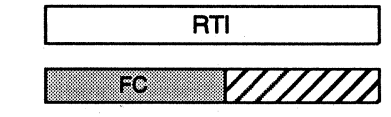
Description Return from interrupts/exceptions by pulling the return address off the system stack into the PC, then pulling the previous status data off the system stack into the ST, and then enabling nonmaskable interrupts.

This instruction is designed to be the return mechanism for peripheral interrupts, TRAPs, or illegal opcodes and their associated exception handling software. The PC must be decremented because interrupts/exceptions leave on the stack a PC value that points two words (four bytes) beyond the address of the next executable instruction in the interrupted stream. This effect is due to the pipeline prefetch of the CPU. The return address is a *word address* that is transformed to a 17-bit physical memory address, via the program counter, by overlaying data bits 0 to 15 onto address lines A16–A1 and forcing line A0 to a 0.

Example

```
RETURN          RTI     ; Return to point of program
                        ; flow when the interrupt
                        ; occurred.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011
2	SP - 2	(RtnA+4) ÷ 2	1011
3	prevA	(prevA)	1011
4	RtnA	IW	1001
5	SP - 4	old ST	1011
6	RtnA + 2	IEW	1000

Syntax **RTS**

Execution
(SP) – 2 → (SP)
((SP)) → (PC)

Modes Supported *Operand not necessary for RTS*

Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description Return from a subroutine by pulling the return address off the system stack into the PC. RTS uses the return linkage created by the CALL and normally is the final instruction of a subroutine entered through the CALL instruction.

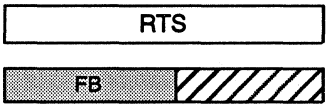
CALL and RTS work together to enter a subroutine and then later return to the instruction following the CALL when the subroutine is exited. The CALL instruction sets up this linkage by placing the PC value (a value that points to the instruction following the CALL) onto the stack before the subroutine is entered.

The return address is a *word address* that is transformed to a 17-bit physical memory address, via the program counter, by overlaying data bits 0 – 15 onto address lines A1 – A16 and forcing A0 to 0. This is illustrated in the CALL/RTS example in Figure 5–4 on page 5-35.

Example

```
Return_1    RTS          ; Return to the instruction
                         ;  immediately following the
                         ;  subroutine call.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}\,\overline{b}\,\overline{f}\,\overline{d}$
1	prevA	(prevA)	1011
2	SP - 2	RtnA + 2	1011
3	RtnA	IW	1001
4	RtnA + 2	IEW	1000

Syntax **SBB**

Execution (destination) – (source) – carry-bit value → (destination)

Modes Supported *Rs,Rd*
**disp16[Rs],Rd*

Status Bits

Z set if the result is zero; unchanged otherwise
N equals the most significant bit of the result
C set if an unsigned underflow occurred; cleared otherwise
V set if a twos complement underflow occurred; cleared otherwise

Description Subtract the contents of the source operand, less the value of C[[ST]], from the destination register.

This instruction is designed to aid 32-bit subtraction. A SUB will subtract the least significant words, and then a following SBB will subtract the most significant words. Since the SBB instruction recognizes a previous underflow/borrow (C[[ST]]), the SUB and SBB instructions must be sequential.

SBB handles Z[[ST]] correctly for 32-bit subtraction. The Z[[ST]] bit is set if and only if the previous operation (typically a SUB) set it. Therefore, all status bits will reflect a 32-bit result after a SUB/SBB sequence of instructions is executed.

Example

```
label   sbb      ZR,R2     ; Subtract the carry bit value
                           ;  from R2. This is a conditional
                           ;  decrement of R2 depending
                           ;  contents of carry bit.
        sbb      R5,R3     ; Subtract R5 value minus carry
                           ;  bit from R3. Result to R3.
        sbb *10h[ZR],r1    ; Subtract 10h minus the carry
                           ;  bit from R1. Result to R1.
```

Instruction Execution Detail

SBB *Rs,Rd*			SBB **disp16[Rs],Rd*		
8C(w)	*Rs*	*Rd*	8D(w)	*Rs*	*Rd*
			disp16		

Cy	Address	Data	$\overline{w}b\overline{i}\overline{d}$	Address	Data	$\overline{w}b\overline{i}\overline{d}$
1	OpA + 4	IEW	1000	OpA + 4	IW	1001
2				disp + Rs	(disp + Rs)	1011
3				OpA + 6	IEW	1000

Syntax **SBIT0{B}**

Execution 0 (zero value) → (bit in destination) (bit number specified in source)

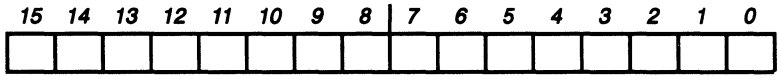
Modes Supported

#imm4,Rd	*(word only)*
*#imm4,*disp16[Rd]*	*(byte only)*
Rs,Rd	*(word only)*
*Rs,*disp16[Rd]*	*(byte only)*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description Clear to 0 a specified bit in the destination. The source value (0–7 for byte, 0–15 for word) specifies which bit to clear in the destination, numbered as shown:



Note that a zero (not the letter O) follows SBIT in the mnemonic. The source value is contained in the 4 least significant bits of a register or in bits 4–7 of the instruction word when an immediate value. If the bit designation value for a byte is in the range 8–15, the instruction performs a *read, no-modify, write* sequence.

Examples Clear bit 4 of register R7: the first example demonstrates an immediate value, and the second demonstrates a register as a source.

```
LABEL     SBIT0    #4,R7    ; Clear 5th bit from right
            ;or
          MOV      #4,R6    ; Immediate bit value to R6
          SBIT0    R6,R7    ; Clear 5th bit from right
```

Instruction Execution Detail

SBIT0 *#imm4,Rd*	SBIT0 *Rs,Rd*	SBIT0B *#imm4,*disp16[Rd]*	SBIT0B *Rs,*disp16[Rd]*
92 (w) *imm4* *Rd*	E2 (w) *Rs* *Rd*	93 (b) *imm4* *Rd* *disp16*	E3 (b) *Rs* *Rd* *disp16*

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2				OpA + 4	IEW	1 0 0 0	disp + Rd	(disp + Rd)	1 1 1 1	prevA	(prevA)	1 0 1 1
3							OpA + 6	IW	1 0 1 0	disp + Rd	(disp + Rd)	1 1 1 1
4							disp + Rd	result	0 1 1 1	OpA + 4	IW	1 0 0 1
5							OpA + 6	IEW	1 0 0 0	disp + Rd	result	0 1 1 1
6										OpA + 6	IEW	1 0 0 0

Syntax **SBIT1{B}**

Execution 1 → (bit in destination) (bit number specified by source)

Modes Supported

#imm4,Rd	*(word only)*
*#imm4,*disp16[Rd]*	*(byte only)*
Rs,Rd	*(word only)*
*Rs,*disp16[Rd]*	*(byte only)*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description Set to 1 a specified bit in the destination. The source value (0–7 for byte, 0–15 for word) specifies which bit to set in the destination, numbered the same as for the SBIT0{B} instruction.

The source value is contained in the 4 least significant bits of a register or in bits 4–7 of the instruction word when an immediate value. If the bit designation value for a byte is in the range 8–15, the instruction performs a *read, no-modify, write* sequence.

Examples Set to 1 the sign bit for the (word) value in register R7:

```
LABEL     SBIT1     #15,R7
```

Set to 1 the sign bit for the (byte) value in address 0701h:

```
LABEL     SBIT1B    #7,&701h
```

Instruction Execution Detail

SBIT1 # *imm4,Rd*	SBIT1 *Rs,Rd*	SBIT1B *#imm4,*disp16[Rd]*	SBIT1B *Rs,*disp16[Rd]*
90 (w) | *imm4* | *Rd*	E0 (w) | *Rs* | *Rd*	91 (b) | *imm4* | *Rd* *disp16*	E1 (b) | *Rs* | *Rd* *disp16*

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2				OpA + 4	IEW	1 0 0 0	disp + Rd	(disp + Rd)	1 1 1 1	prevA	(prevA)	1 0 1 1
3							OpA + 4	IW	1 0 0 1	disp + Rd	(disp + Rd)	1 1 1 1
4							disp + Rd	result	0 1 1 1	OpA + 4	IW	1 0 0 1
5							OpA + 6	IEW	1 0 0 0	dis + Rd	result	0 1 1 1
6										OpA + 6	IEW	1 0 0 0

Syntax **SHL{L}**

Execution shift left the destination register(s) by source count → (destination register(s))

Modes Supported

#imm4,Rd	*(word)*
#imm4,IM:Rd	*(longword)*
Rs,Rd	*(word)*
Rs,IM:Rd	*(longword)*

Status Bits

Z set if the result is zero, cleared otherwise

N equals an XOR (exclusive OR) of the V[[ST]] bit after a shift with the destination's most significant bit before shifting

C set if a one is *ever* shifted out of the register; cleared otherwise

V set if the most significant bit of the register *ever* changes during the shift; cleared otherwise

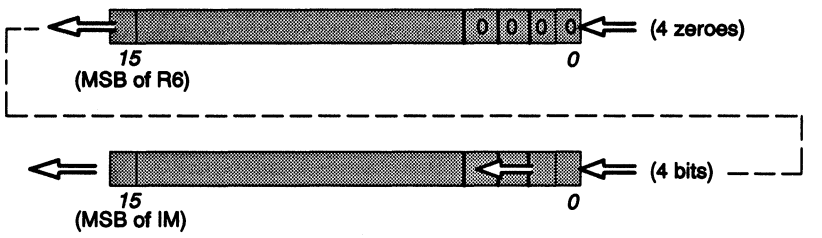
Description Arithmetically shift left the destination register's signed contents by the number of bit positions specified in the source operand. Shift zero(es) into the vacated least significant bit(s). The four least significant bits of the source operand contain the shift count (range of 0–15).

For **immediate** shifts, a source operand value of 0001_2 to 1111_2 indicates a shift count of 1 to 15; a source operand value of 0000_2 indicates a shift count of 16. If an immediate shift count of more than four bits (more than 15) is specified, the least-significant four hexadecimal bits (of the value specified) are assembled.

The following depicts the movement within the destination register:

The illustration below depicts a longword shift using the concatenation of IM and Rd:

```
SHL  R5,IM:R6          ; R5 = 4 (shift count)
```

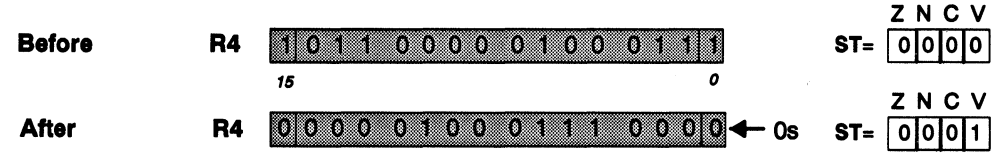


This instruction performs a mathematically correct multiply of the destination contents by a power of 2 (2^1–2^{16}). Another way to view execution is as a series of identical additions of the destination contents to itself — one addition (or doubling of itself) for each bit shifted. All of the status bits are "sticky" (the value remains the same after each shift). If any normal ADD operation overflow conditions occur during the ADD repetitions, this will be reflected in the C[ST] or V[ST] condition code bits. The N[ST] bit is correct for a repetitive add and will always be cleared if a twos-complement overflow occurs on a negative number.

Status bits are set with respect to the size of the word shifted (16 or 32 bits). Longword shifts always use the IM as the most significant word of the 32-bit object. The result of SHL (source),IM:IM is undefined.

Example

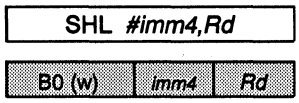
```
SHL     #4,R4    ;shift R4 bits to left 4 bits
```

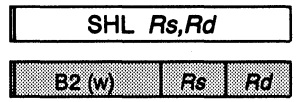


The N[ST] bit reflects an XOR of the sign bit before execution (a 1) and the V[ST] after execution (a 1 because the sign changed at least once).

Instruction Execution Detail

Word Instructions (2 + *n* cycles)

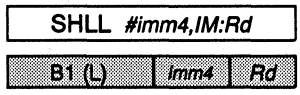


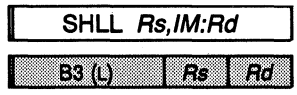


Cycle/ Period†	Address	Data	w b i d
1	prevA	(prevA)	1 0 1 1
n (repeat)	prevA	(prevA)	1 0 1 1
n + 2	OpA + 4	IEW	1 0 0 0

Cycle/ Period†	Address	Data	w b i d
1, 2	prevA	(prevA)	1 0 1 1
n (repeat)	prevA	(prevA)	1 0 1 1
n + 2	OpA + 4	IEW	1 0 0 0
	Total cycles: *n* + 3		

Longword Instructions (2 + *2n* cycles)





Cycle/ Period†	Address	Data	w b i d
1	prevA	(prevA)	1 0 1 1
2*n* (repeat)	prevA	(prevA)	1 0 1 1
2 + 2*n*	OpA + 4	IEW	1 0 0 0

Cycle/ Period†	Address	Data	w b i d
1, 2	prevA	(prevA)	1 0 1 1
2*n* (repeat)	prevA	(prevA)	1 0 1 1
2 + 2*n*	OpA + 4	IEW	1 0 0 0
	Total cycles: 2*n* + 3		

† A single number represents a *given cycle*; an expression of *n* represents a *cycle* or *period of cycles*, depending on the n^{th} number of shifts or repeats.

Syntax **SHL4**

Execution source shifted four bits to left → (destination)

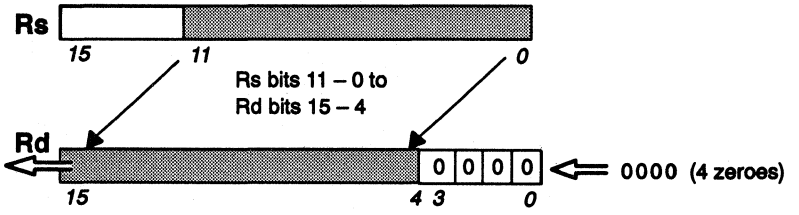
Mode Supported *Rs,Rd*

Status Bits

Z set if the result is zero; otherwise cleared
N equals the result's most significant bit
C unchanged
V unchanged

Description Logically left-shift the source register's contents four bit positions. Shift zero(es) into the four least significant bits. Place the results of the shift into the destination register. Execution changes *only Rd's contents*. This instruction effectively multiplies the contents of Rs by 16 and places the unsigned product in Rd.

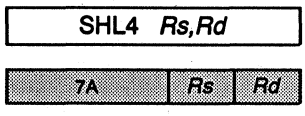
This can also be represented as shifting four zeroes into Rd and copying bits 11 – 0 of Rs into bits 15 – 4 of Rd as shown below:



Example

```
Label   SHL4    R3,R11  ; Logically shift R3 left 4
                        ; bits then load the result
                        ; into R11. Effectively this
                        ; is a multiply of R3 by 16 with
                        ; the results placed in R11.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d
1	prevA	(prevA)	1011
2	OpA + 4	IEW	1000

Syntax **SHL8**

Execution source shifted eight bits to left → (destination)

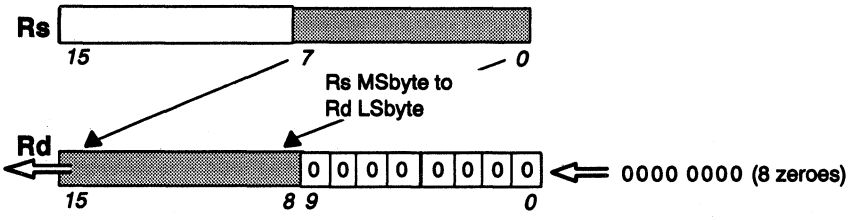
Mode Supported *Rs,Rd*

Status Bits

Z set if the result is zero; otherwise, cleared
N cleared
C unchanged
V unchanged

Description Logically left-shift the source register's contents eight bit positions. Shift zero(es) into the eight least significant bits. Place the results of the shift into the destination register. Execution changes *only Rd's contents.* This instruction effectively multiplies the contents of Rs by 256 and places the unsigned product in Rd.

This can also be represented as shifting eight zeroes into Rd and copying the LSbyte of the Rs into the MSbyte of Rd as shown below:

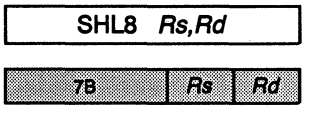


Essentially, the least significant byte of Rs (before shift) is placed in the most significant byte of Rd with the least significant byte of Rd cleared.

Example

```
LABEL   SHL8  R6,R5     ; Logically shift R6 left 8 bits
                        ;  then load the result into R5.
                        ;  Effectively this is a
                        ;  multiply of R6 by 256 with
                        ;  the result placed in R5.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{t}$ d
1	prevA	(prevA)	1011
2	OpA + 4	IEW	1000

Syntax **SHR8**

Execution source shifted eight bits to right → (destination)

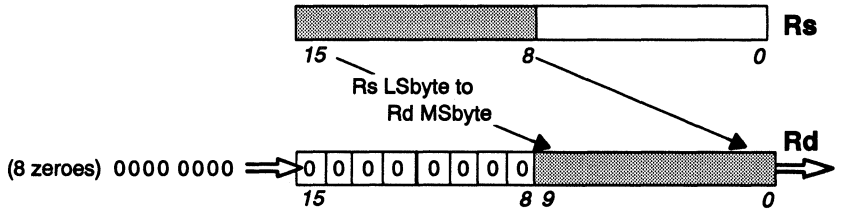
Mode Supported *Rs,Rd*

Status Bits

Z set if the result is zero; otherwise, cleared
N cleared
C unchanged
V unchanged

Description Logically right-shift the source register's contents eight bit positions. Shift zero(es) into the register's most significant eight bits. Place the results of the shift into the destination register. Execution changes *only Rd's contents*. This instruction effectively divides the contents of Rs by 256 and places the unsigned quotient in Rd.

This can also be represented as shifting eight zeroes into Rd and copying the MSbyte of the Rs into the LSbyte of Rd as shown below:



Note that the most significant byte of Rs (before shift) is placed in the least significant byte of Rd with the least significant byte of Rd cleared.

Example

```
LABEL   SHR8  R6,R5     ; Logically shift R6 right 8 bits
                        ;  then load the result into R5.
                        ;  Effectively this is a
                        ;  divide of R6 by 256 with
                        ;  result placed in R5.
```

Instruction Execution Detail

SHR8 *Rs,Rd*		
7C	*Rs*	*Rd*

Cy	Address	Data	$\overline{w}$ b f $\overline{d}$
1	prevA	(prevA)	1011
2	OpA + 4	IEW	1000

Syntax **STBIT{B}**

Execution ones complement the selected destination bit → (Z⟦ST⟧)
(C⟦ST⟧) → (selected destination bit)

Modes Supported

*#imm4,*disp16[Rd]*	*(byte only)*
Rs,Rd	*(word only)*
*Rs,*disp16[Rd]*	*(byte only)*
#imm4,Rd	*(word only)*

Status Bits

Z set if bit tested is 0; cleared if bit is 1
N unchanged
C unchanged
V unchanged

Description

The 3- or 4-bit *source* value is the number of the *destination bit* to be manipulated (bit-number range of 0–7 or 0–15, depending on byte or word instruction). (Bits are numbered as shown for the SBIT0 instruction on page 5-89.) Execution sequence is as follows:

1. Read the value of the selected destination bit and store the ones complement of this value in the Z bit of the status register.
2. Store the status register's C bit value into the selected bit position in the destination.

This sequence provides a means to check a semaphore in memory. And, if an "available" indication is found, the semaphore is then set to the needed value in order to gain control of a function (such as a bus, as shown in examples on next page, which use SBIT1 and SBIT0 to set up the Z⟦ST⟧ value).

Also, since the Z⟦ST⟧ receives the ones complement of the bit value, a zero in the bit tested would cause a branch by the instruction BEQ.

The source value is stored in bits 7–4 of the opcode or the least significant bits of a register.

Useful single bit values are 0–7 for byte (destination a memory address *only*) and 0–15 for word with destination in a register. When the bit-selection value for *byte is 8–15*, a *read, no modify, write* sequence executes and the Z⟦ST⟧ bit is left equal to 1. Bits are numbered as shown for the SBIT0 instruction on page 5-89.

Example

This instruction makes possible a semaphore test operation by preceding the STBIT instruction with a SBIT1 or SBIT0 that sets or clears the C⟦ST⟧ bit. In the following examples, address 1000h is a dedicated word of 16 semaphores. A 1 at bit 2 of the address indicates that a bus is busy. The following code polls the semaphore for a 0, indicating that the bus is available:

Wait for Zero at Semaphore (Loop Until a Zero Is Found at Bit 2 of 1000h):

```
LOOP    SBIT1     #CARRY,ST      ; Set CARRY bit = 1
        STBITB    #2,*1000h[ZR] ; Is semaphore 0 yet?
        BNZ       LOOP           ; Loop until bit #2=0
                  •
                  •
                  •
;     . . .  when bit #2 of 1000h = zero, STBIT sets
;       the bit to one to hold the bus; now enter
;       bus service routine and clear semaphore
;       upon exit.

                  •
                  •
                  •
        SBIT0     #2,*1000h[ZR]
; Exit, clear semaphore
```

When the semaphore becomes a 0 (bus available), the STBIT instruction automatically sets it to a 1 (transfers the set C[[ST]] bit to the semaphore) to maintain bus possession by the new owner. When the bus is needed no longer, set the semaphore to 0 before exiting.

The bus-busy indicator could be the opposite of that above: a 0, with a loop needed to find a 1. In this case, the C[[ST]] bit is cleared (SBIT0), and the conditional branch loops on finding a 1 (inverted semaphore value).

Wait for One at a Semaphore (Loop Until a One Is Found at Bit 2 of 1000h):

```
LOOP    SBIT0     #CARRY,ST      ; Set CARRY bit = 0
        STBITB    #2,#1000h[ZR]  ; Is semaphore 1 yet?
        BEQ       LOOP           Loop until bit #2=1
                  •
                  •
                  •
;   . . .  when bit #2 of 1000h = one, the bus can be
;       obtained; enter bus service routine then
;       set semaphore upon exit.;
                  •
                  •
                  •
        SBIT1     #2,#1000h[ZR]  ; Exit, set semaphore
```

Instruction Execution Detail

STBIT *#imm4,Rd*	STBIT *Rs,Rd*	STBITB *#imm4,*disp16[Rd]*	STBITB *Rs,*disp16[Rd]*
96 (w) | *imm4* | Rd	E6 (w) | *Rs* | *Rd*	97 (b) | *imm4* | *Rd* *disp16*	E7 (b) | *Rs* | *Rd* *disp16*

Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
2	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1	disp + Rd	(disp + Rd)	1 1 1 1	prevA	(prevA)	1 0 1 1
3				OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1	disp + Rd	(disp + Rd)	1 1 1 1
4							OpA + 4	IW	1 0 0 1	prevA	(prevA)	1 0 1 1
5							disp + Rd	result	0 1 1 1	OpA + 4	IW	1 0 0 1
6							OpA + 6	IEW	1 0 0 0	disp + Rd	result	0 1 1 1
7										OpA + 6	IEW	1 0 0 0

Syntax **STEA**

Execution *disp16* + (*Rs*) → (Rd)
(Rd) + 2 → (Rd)

Mode Supported ***disp16[Rs],*Rd+*

Status Bits

Z	unchanged
N	unchanged
C	unchanged
V	unchanged

Description Sum the $disp_{16}$ value and the contents of *Rs*, and indirectly store this in the address pointed to by the destination register. Then increment the destination register contents by 2.

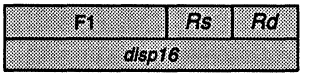
Example Given: R4 = 0002h and R5 = 8000h. The following code moves the value 6002h (the sum of the 6000h displacement and (R4)) into memory address 8000h and increments R5 to the value 8002h.

```
Label       STEA    *6000h[r4],*r5+
```

	Before	After
R4	0 0 0 2	0 0 0 2
R5	8 0 0 0	8 0 0 2
8000h	N/A	6 0 0 2

Instruction Execution Detail

STEA * *disp16[Rs],*Rd+*



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	OpA + 4	IW	1 0 0 1
2	Rd	disp + Rs	0 0 1 1
3	OpA + 6	IEW	1 0 0 0

Syntax **STRI**

Execution (ST) → (*Rd*)
imm4 → (4 LS bits of the ST)

Mode Supported *#imm4,Rd*

Status Bits
Z cleared
N cleared
C cleared
V cleared

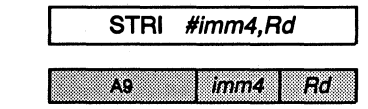
Description Store the contents of the ST into *Rd*. Then copy the three LSBs of *imm4* into the **three interrupt-level bits** of the ST and clear its Z, N, C, and V bits.

The most significant byte of the ST is undefined because of ST reserved bits (these bits are undefined when read and don't retain data when written to).

Example

```
Label   STRI    #01h,R2     ; Store the ST into R2, then
                            ; set the 2 LS bits of the
                            ; ST to 01b (01 binary).
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2	OpA + 4	IEW	1 0 0 0

Syntax **SUB{B}**

Execution (destination) – (source) → (destination)

Modes Supported
Rs,Rd
#imm16,Rd
**disp16[Rs],Rd*
*Rs,*disp16[Rd]*

Status Bits
Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C set if an unsigned underflow occurred; cleared otherwise
V set if a twos complement underflow occurred; cleared otherwise

Description Subtract the contents of the source operand from the destination operand. Source contents are left unchanged.

For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will be either 00h for C[[ST]]=0 or FFh for C[[ST]]=1. Nonregister destinations receive the least significant byte of the result, while registers receive the entire word.

Status bits are set with respect to the size (byte or word) of the operation.

Example

```
label   SUB    R5,R8      ; Subtract contents of R5 from
                          ;  R8. Store result in R8.
sbtrct  SUB    R10,&LAST  ; Subtract contents of R10 from
                          ;  the value in location LAST.
                          ;  Leave results in LAST.
        SUBB   #5,R2      ; Subtract 5 from R2 contents,
                          ;  and set MSbyte of R2 = 00h.
```

Instruction Execution Detail

SUB{B} *Rs,Rd*	SUB{B} *#imm16,Rd*	SUB{B} **disp16[Rs],Rd*	SUB{B} *Rs,*disp16[Rd]*
38(w) 39(b) | *Rs* | *Rd*	3C(w) 3D(b) | /// | *Rd* *imm16*	3E(w) 3F(b) | *Rs* | *Rd* *disp16*	3A(w) 3B(b) | *Rs* | *Rd* *disp16*

Cy	Address	Data	w̄ b f̄ d̄	Address	Data	w̄ b f̄ d̄	Address	Data	w̄ b f̄ d̄	Address	Data	w̄ b f̄ d̄
1	OpA + 4	IEW	1000	OpA + 4	IW	1001	OpA + 4	IW	1001	OpA + 4	IW	1001
2				OpA + 6	IEW	1000	disp + Rs	(disp + Rs)	1S11	disp + Rd	(disp + Rd)	1S11
3							OpA + 6	IEW	1000	prevA	(prevA)	1011
4										disp + Rd	result	0S11
5										OpA + 6	IEW	1000

Syntax **SUBQ{B}**

Execution (destination) – immediate data → (destination)

Modes Supported
#imm4,Rd
*#imm4,*disp16[Rd]*

Status Bits
Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C set if an unsigned underflow occurred; cleared otherwise
V set if a twos complement underflow occurred; cleared otherwise

Description Subtract the quick immediate data from the contents of the destination operand. Quick immediate data consists of a 4-bit value of 0–15, contained within the instruction word, which has been zero-extended to the correct data object size. The SUBQ *#data,Rd* instruction takes one cycle to execute, while the SUB equivalent takes two cycles.

For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will be either 00h for C[[ST]]=0 or FFh for C[[ST]]=1. Nonregister destinations receive the least significant byte of the result, while registers receive the entire word.

Status bits are set with respect to the size (byte or word) of the operation.

Example

```
label   SUBQB   #7,IM     ; Subtract 7 from the IM.
                          ;
        SUBQ #FIVE,&FINISH ; Subtract FIVE's value
                          ; from location FINISH
```

Instruction Execution Detail

SUBQ{B} *#imm4,Rd*	SUBQ{B} *#imm4,*disp16[Rd]*

86(w) 87(b)	*imm4*	*Rd*	88(w) 89(b)	*imm4*	*Rd*
			disp16		

Cy	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
2				disp + Rd	(disp + Rd)	1 S 1 1
3				OpA + 4	IW	1 0 0 1
4				disp + Rd	result	0 S 1 1
5				OpA + 6	IEW	1 0 0 0

Syntax **SUBR{B}**

Execution (Register B) – (Register A) → (Register A)

Mode Supported *RA,RB*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C set if an unsigned underflow occurred; cleared otherwise
V set if a twos complement underflow occurred; cleared otherwise

Description Subtract the contents of register A from the contents of register B and place the result back into register A. (Subtract source from destination, but place result back into source.)

Note that when RB is the ZR, the instruction is equivalent to the instruction COMPL *RA*.

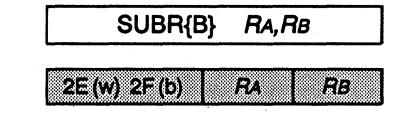
For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will be either 00h for C[[ST]]=0 or FFh for C[[ST]]=1.

Status bits are set with respect to the size (byte or word) of the operation.

Example

```
label   SUBR    R3,R7   ; Subtract R3 from R7,
                        ;  Store result in R3.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}\,\overline{b}\,\overline{f}\,\overline{d}$
1	OpA + 4	IEW	1000

Syntax **SWAPB**

Execution (*Rs* (LSbyte)) → (*Rd* (MSbyte))
(*Rs* (MSbyte)) → (*Rd* (LSbyte))

Mode Supported *Rs,Rd*

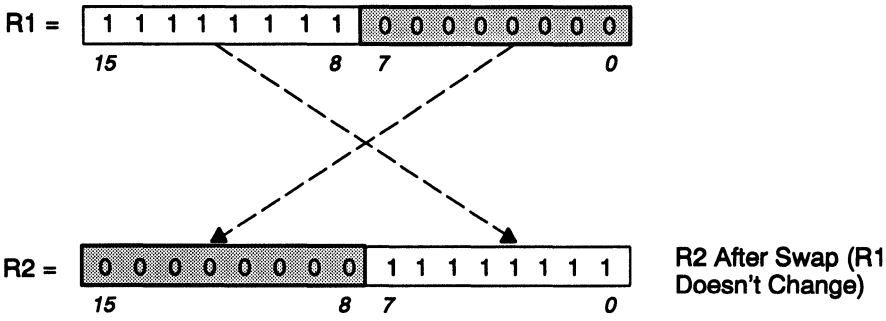
Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C unchanged
V cleared

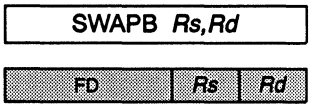
Description Copy (swap) the source register's most significant byte and its least significant byte with the opposite bytes of the destination register. The source register remains unchanged.

Example

```
LABEL    SWAPB    R1,R2
```



Instruction Execution Detail



C	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	Rs_LSbyte	0 1 1 1
2	prevA	Rs_MSbyte	0 1 1 1
3	OpA + 4	IEW	1 0 0 0

Syntax **TBIT0**

Execution IF [(mask ≠ 0) **and** (mask ANDed to destination = 0)],
THEN 1 → (Z[[ST]])
ELSE 0 → (Z[[ST]])

Mode Supported *#imm8*(mask),*&addr16* (The **&** operator **must** be included as shown. The # operator in front of *imm* is optional.)

Status Bits
Z set if tested bits are cleared; otherwise, a zero
N unchanged
C unchanged
V unchanged

Description This is a byte instruction only.

For each logical 1 bit in the source mask, test the corresponding bit in the destination-address *byte*. If *all* specified destination bits are 0s, place a 1 in the Z bit of the status register. Otherwise, set the Z bit to 0. *Only the 1 bits in the mask are ANDed to set the Z bit.* If the source mask is all zeroes (00h), no bits are tested and bit Z[[ST]] is cleared.

The destination byte is always in the first 64K bytes of memory and is addressed by a 16-bit value (address line A16 = 0).

This instruction is designed to be followed by a BEQ (branch if equal) or BNE (branch not equal) instruction to form, respectively, a *branch on multiple bits clear* or *branch on multiple bits not clear* operation.

Example While moving a block of bytes from one memory area to another, check each byte for all zeroes in bits 0, 1, 2, and 4. If all are zeroes, move the next byte and continue. If not all ones, do a bit check routine before moving the next byte.

```
START    MOVB    *R7,*R8     ;Bring in (next) byte to check
         MOVB    *R8+,4000h  ;Place in memory for bit check
         TBIT0   0Bh,&4000h  ;Are bits 0, 1, 3 cleared?
         BEQ     START       ; If bits are clear, move next byte
                             ; If not clear, do bit check
BIT_CHK          .          ; Start of bit checking
                 .
                 .
         JMP     START       ; After check, get next byte
```

Instruction Execution Detail

TBIT0 imm_8,&addr

F4 (b)	imm_8
addr	

Cy	Address	Data	$\overline{w}$ b f $\overline{d}$
1	addr	(addr)	1 1 1 1
2	OpA + 4	IW	1 0 0 1
3	OpA + 6	IEW	1 0 0 0

Syntax **TBIT1**

Execution IF [(mask ≠ 0) *and* (mask_ones ANDed to inverted destination = 0)],
THEN 1 → (Z[[ST]])
ELSE 0 → (Z[[ST]])

Mode Supported *#imm8*(mask),&*addr16* (The **&** operator **must** be included as shown. The # operator in front of *imm* is optional.)

Status Bits

Z set if tested bits are set (ones); otherwise, a zero
N unchanged
C unchanged
V unchanged

Description This is a byte instruction only.

For each logical 1 bit in the source mask, test the corresponding bit in the destination-address *byte*. If *all* specified destination bits are 1s, place a 1 in the Z bit of the status register. Otherwise, set the Z bit to 0. *Only bits corresponding to the 1 bits in the mask are tested to set the Z bit.* If the source mask is all zeroes (00h), no bits are tested and bit Z[[ST]] is cleared.

The destination byte is always in the first 64K bytes of memory and is addressed by a 16-bit value (address line A16 = 0).

This instruction is designed to be followed by a BEQ (branch if equal) or BNE (branch not equal) instruction to form, respectively, a *branch on multiple bits set* or *branch on multiple bits not set* operation.

Example While moving a block of bytes from one memory area to another, check each byte for all 1s in bits 4–7. If all are 1s, move next byte and continue. If not all 1s, do a bit check routine before moving the next byte.

```
START    MOVB    *R7+,*R8       ; Bring in (next) byte
         MOVB    *R8+,4000h     ; Byte to memory
         TBIT1   0F0h,&4000h    ; Are bits 4-7 set?
         BEQ     START          ; If bits set, move next byte
                                ; If not set, do bit check
BIT_CHK          .             ; Start of bit check
                 .
                 .
         JMP     START          ; After check, get next byte
```

Instruction Execution Detail

TBIT1 *imm8,&addr*	
F5 (b)	*uimm8*
addr	

Cy	Address	Data	$\overline{w}\,b\,\overline{i}\,\overline{d}$
1	addr	(addr)	1 1 1 1
2	OpA + 4	IW	1 0 0 1
3	OpA + 6	IEW	1 0 0 0

Syntax **TBLU{B}**

Execution

(*Rs*(MSbyte)) + (Rd) → (IM)
(*Rs*(MSbyte)) + (Rd) + size → (Rd)
IF IM > RD
THEN
    Rs(LSbyte) × (IM – Rd) + 80h → TEMP (8 bits × 16 bits → 24 bits + 80h)
TEMP ÷ 256 → Rd
    IM – Rd → Rd
ELSE
    Rs(LSbyte) × (IM – Rd) + 80h → TEMP (8 bits × 16 bits → 24 bits + 80h)
TEMP ÷ 256 → Rd
    IM + Rd → Rd

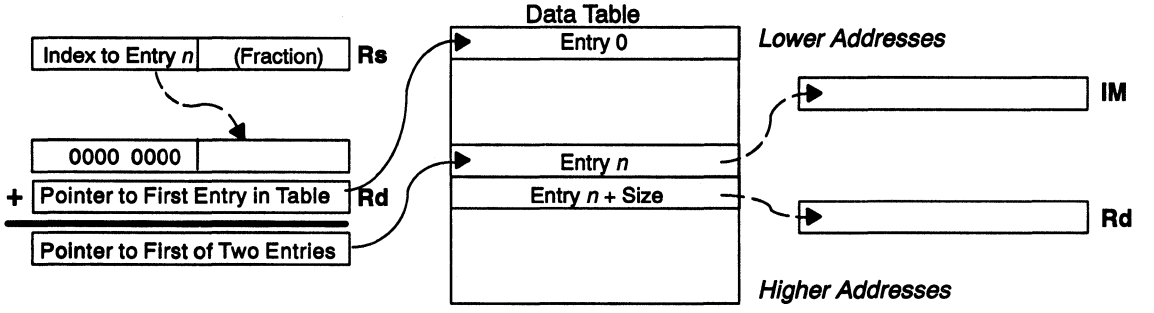
Mode Supported *Rs,IM:Rd*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C cleared
V cleared

Description Look up two consecutive values in a table of unsigned data, referenced by Rd, and perform a rounded straight-line interpolation between them, according to the interpolation fraction in Rs. The result is rounded to fit the byte/word size of the instruction and then placed in Rd.

The 16-bit address in Rd points to the first entry of the data table. This table is indexed by normalizing the most significant byte of Rs and adding it to Rd. This sum yields the address of the first of two consecutive entries in the table for which interpolation is to be computed. The two table entries are then read into IM and Rd respectively, as illustrated below.



Note: Dotted line shows value moved; solid line indicates location pointed to.

Notes: Considerations for >64K Bytes and Effect of Byte Size on Registers

1. The calculated table pointer in Rd is a 16-bit value that can address only the first 64K bytes of memory (A16 = 0). Attempts to generate a result that points beyond the first 64K bytes of memory will wrap around to the beginning of the first 64K bytes of memory.
2. If the instruction size is byte, the most significant bytes of IM and Rd will be cleared when the table entries are read.

The interpolation fraction is held in the least significant byte of Rs and has its radix point between bits 7 and 8. The most significant byte of Rs is ignored during multiplication. The contents of Rs are left unchanged.

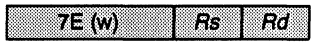
The internal multiply is 8 x 16 where the 8-bit value is the fraction and the 16-bit value is the appropriate difference between the two table entries read into IM and Rd. The product is a 24-bit fixed-point value with the integer portion in bits 8–23 and the fraction in bits 0–7. This intermediate product is rounded up to word value in bits 8–23 by adding 000080h. This rounded result is then combined with IM, yielding the final interpolated result, which is placed into Rd.

The fractional portion of the intermediate product is lost. The operand combination TBLU{B} Rs,IM:IM will always generate a result of 0000h in IM. **Undefined execution** results in the combination TBLU{B} Rs,IM:ZR; thus, it must be avoided.

Status bits are set with respect to the size (byte/word) of the operation.

Instruction Execution Detail

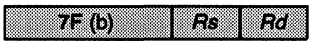
TBLU *Rs,IM:Rd*



Cy	Entry 1 ≤ Entry 2			Entry 1 > Entry 2		
	Address	Data	w b i d	Address	Data	w b i d
1,2	prevA	(prevA)	0 0 1 1	prevA	(prevA)	0 0 1 1
3	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
4	Rd + Rs_MS_byte	(Rd + Rs_MS_byte)	1 0 1 1	Rd + Rs_MS_byte	(Rd + Rs_MS_byte)	1 0 1 1
5	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
6	Rd + Rs_MS_byte + 2	(Rd + Rs_MS_byte + 2)	1 0 1 1	Rd + Rs_MS_byte + 2	(Rd + Rs_MS_byte + 2)	1 0 1 1
7–13	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
14	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
15	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
16				OpA + 4	IEW	1 0 0 0

Instruction Execution Detail (Concluded)

TBLUB *Rs,IM:Rd*



Cy	Entry 1 ≤ Entry 2			Entry 1 > Entry 2		
	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1, 2	prevA	(prevA)	0 0 1 1	prevA	(prevA)	0 0 1 1
3	Rd + Rs_MS_byte	(Rd + Rs_MS_byte)	1 1 1 1	Rd + Rs_MS_byte	(Rd + Rs_MS_byte)	1 1 1 1
4	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
5	Rd + Rs_MS_byte + 1	(Rd + Rs_MS_byte + 1)	1 1 1 1	Rd + Rs_MS_byte + 1	(Rd + Rs_MS_byte + 1)	1 1 1 1
6	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
7–13	prevA	(prevA)	1 0 1 1	prevA	(prevA)	1 0 1 1
14	OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
15				OpA + 4	IEW	1 0 0 0

Syntax **TRAP**

Execution

(ST) → ((SP))
(SP) + 2 → (SP)
(PC) + 1 → ((SP))
(SP) + 2 → (SP)
ones complement of enumerator × 2 → vector offset
vector table base addr + vector offset → (PC) *(subroutine address → PC)*
1s → L2–L0[[ST]]

Mode Supported *imm8* *[#imm8 = trap number (0–255); enumerator8 = ones complement of trap number]*

Status Bits

Z unchanged
N unchanged
C unchanged
V unchanged

Description The TRAP instruction operates as a software interrupt or exception. A 256-word trap vector table, located at a vector-table base address, contains the start addresses of each trap subroutine (TRAP 0 being at the lowest address in the table). This is shown graphically in Figure 5–6 on page 5-113.

Note: Five Trap Words Are Reserved

The 'C16 trap vector table contains mask ROM space reserved for **TI use only**—addresses 08036h–08040h, as shown in Figure 5–6 on page 5-113. This reserved area should not be used in your software algorithm, nor should it be used during mask ROM/firmware development.

A summary of the trap exception steps:

1) Push the current ST contents on to the stack; then increment the SP by 2.
2) Add 1 to the PC value and place the result on the system stack (this will point two words beyond the next instruction). Increment the SP by 2.
3) Calculate the vector offset (from the trap vector-table base address) by multiplying a ones complement of the instruction's enumerator by 2. (The enumerator is stored in the LSbyte of opcode as the ones complement of the trap number.)
4) Load the PC with the trap vector-table base address + vector offset (address containing the trap-subroutine start address).
5) Load the PC with the subroutine start address.
6) Load all 1s into the ST's three interrupt level bits (L2–L0)

This instruction replicates a peripheral interrupt. In this manner, it is a software interrupt and requires you to provide an interrupt/exception handler in software. *Use an RTI instruction to return to the interrupted execution flow.*

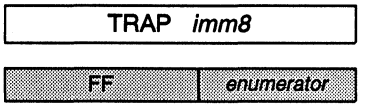
Note: TRAP Enumerator Source

Note that the enumerator value as assembled in the LSbyte of the opcode is the inverse (ones-complement) trap value. For example, TRAP 0 is the instruction word FFFFh (FFh is the enumerator value), TRAP 1 is FFFEh (FEh the enumerator value), and TRAP 255 is FF00h (00h is the enumerator value). (This explains the ones-complement computation in the Execution equation above.) **Traps are further explained** in subsection 3.7.6 on page 3-24.

Example

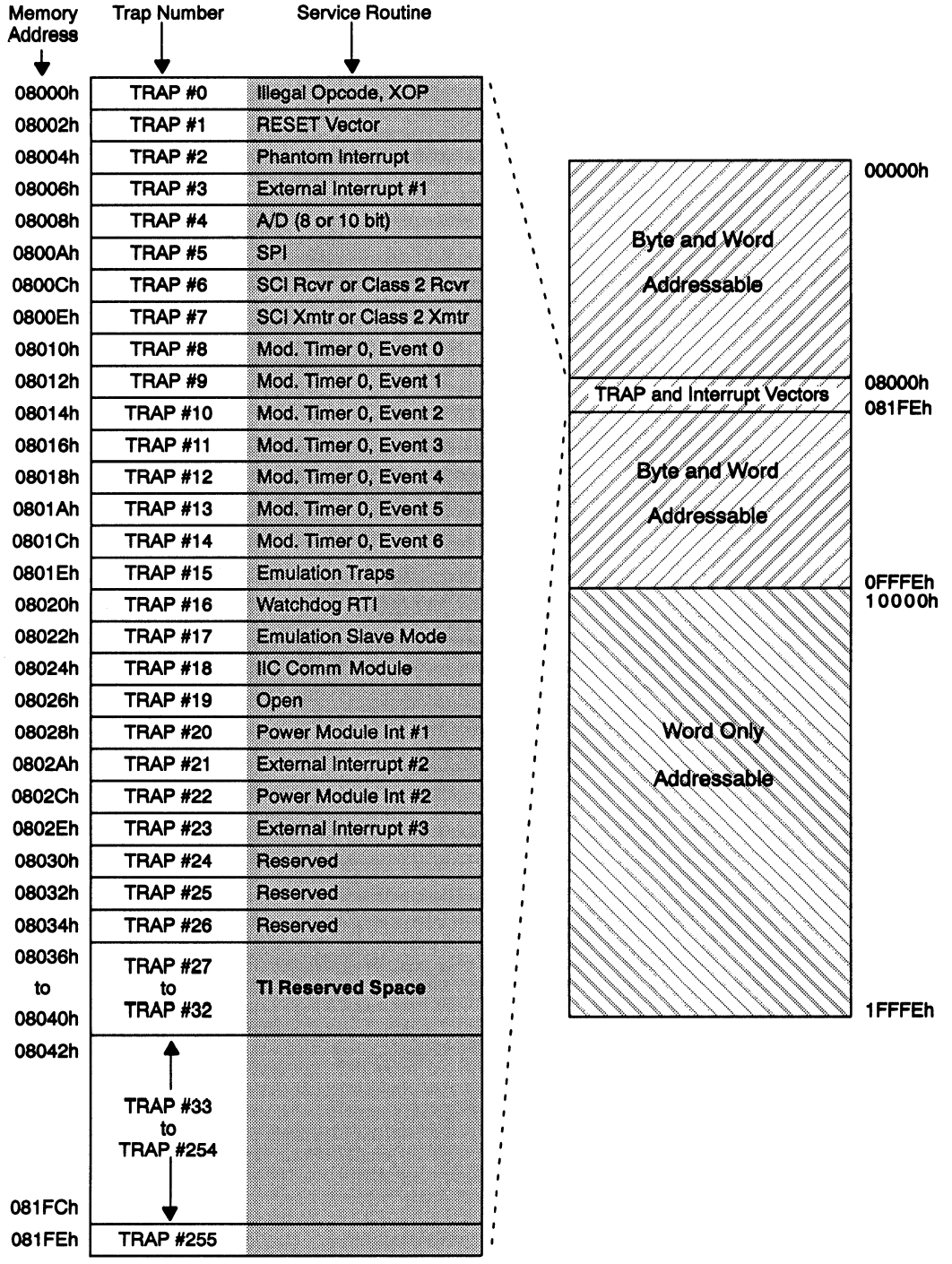
```
Label   TRAP    32      ; Call TRAP 32 vector. Begin
                        ;  execution at the address
                        ;  stored at that location.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	OpA + 4	IEW	1 0 0 1
2	SP	ST	0 0 1 1
3	SP + 2	(Rtn A + 4) ÷ 2	0 0 1 1
4	prevA	(prevA)	1 0 1 1
5	(NOT *enum*) × 2 + vector base_addr	subroutine start address	1 0 1 1
6	subroutine start address × 2	IW	1 0 0 1
7	(subroutine start address × 2) + 2	IEW	1 0 0 0

Figure 5–6. Vector Table for TRAP Instruction



Syntax **TRUNCS{L}**

Execution

IF [valid truncation *not* possible]
THEN one → V[ST]
ENDIF

Modes Supported

Rd	*(word only)*
IM:Rd	*(longword only)*

Status Bits

Z TRUNCS: set if the least significant byte of *Rd* is zero; cleared otherwise
TRUNCSL: set if *Rd* is zero; cleared otherwise

N equals V[ST] XORed with the most significant bit of the original data object

C cleared

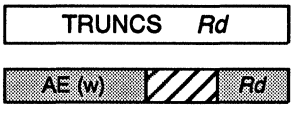
V TRUNCS: set if bits 15 to 7 of *Rd* are not the same; cleared otherwise
TRUNCSL: set if all bits in IM **and** bit 15 of *Rd* are not the same; cleared otherwise

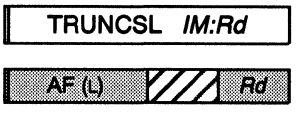
Description

Test the signed data in the register(s) to determine if it is possible to accurately represent the data in the next smaller data object size. If *not* possible, set the V bit in the status register to a one.

Use the BV (branch if overflow set with V[ST] = 1) or BNV (branch if overflow not set with V[ST] = 0) instructions to decide.

Instruction Execution Detail





	bits 7 - 15 are the same			bits 7 - 15 are not the same					
Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1–2	prevA	(prevA)	1011	prevA	(prevA)	1011	prevA	(prevA)	1011
3	OpA + 4	IEW	1000	prevA	(prevA)	1011	prevA	(prevA)	1011
4				OpA + 4	IEW	1000	OpA + 4	IEW	1000

Syntax **TRUNCU**

Execution

IF [valid truncation *not* possible],
THEN, one → C[[ST]]
ENDIF

Mode Supported *Rd*

Status Bits

Z set if the least significant byte of *Rd* is zero; cleared otherwise
N equals the value of bit 7 (most significant bit of the byte result)
C set if bits 8 – 15 of *Rd are not* zero; cleared otherwise
V cleared

Description

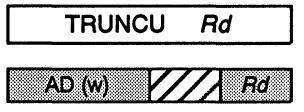
Test the unsigned data word in the destination register to determine if it can be accurately represented as a byte data object. If *not* possible, set the C bit in the status register to a 1.

Use the BC (branch if carry set with C[ST] = 1) or BNC (branch if carry clear with C[ST] = 0) instructions to decide.

Example

```
LABEL   TRUNCU  R2        ; DOES MS BYTE OF R2 = 0?
        BNC     Byte_Val  ; YES, JUMP TO BYTE ROUTINE:
                          ;  OTHERWISE, CONTINUE
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ d
1	prevA	(prevA)	1011
2	OpA + 4	IEW	1000

Syntax **TST{B}** *Synthetic Instruction: Executes as MOV s,ZR*

Execution MOV *s,ZR*

Modes Supported
Rs
**Rs*
**Rs+*
#imm16
**dips16[Rs]*
**–Rs*

Status Bits
Z set if the source is zero; cleared otherwise
N equals the most significant bit of the source
C unchanged
V cleared

Description Test the value of the source operand by moving (copying) it to the ZR (R15). Set the ST bits accordingly. The source value is not changed.

Byte operations test only the least significant byte of a register. Status bits are set with respect to the size (byte or word) of the operation.

Example

```
TSTB    *0A1h[ZR]          ; Check byte address 0A1h.
                           ;  Set status bits on result.
Check   TST    &VALUE      ; Check word location VALUE.
                           ;  Set status bits on result.
```

Instruction Execution Detail

TST{B} *Rs* (MOV{B} Rs,ZR)	TST{B} **Rs* (MOV{B} *Rs,ZR)	TST{B} **Rs+* (MOV{B} *Rs+,ZR)
02(w) 03(b) | *Rs* | 1111_2	0A(w) 0B(b) | *Rs* | 1111_2	12 (w) 13 (b) | *Rs* | 1111_2

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IEW	1 0 0 0	Rs	(Rs)	0 S 1 1	Rs	(Rs)	0 S 1 1
2				OpA + 4	IEW	1 0 0 0	prevA	(prevA)	1 0 1 1
3							OpA + 4	IEW	1 0 0 0

TST{B} *#imm16* (MOV{B} #imm16,ZR)	TST{B} **disp16[Rs]* (MOV{B} *disp16[Rs],[ZR])	TST{B} **–Rs* (MOV{B} *–Rs,ZR)
1A (w) 1B (b) | *Rs* | 1111_2	22 (w) 23 (b) | *Rs* | 1111_2	2A (w) 2B (b) | *Rs* | 1111_2
Imm16	*disp16*	

Cy	Address	Data	w b f d	Address	Data	w b f d	Address	Data	w b f d
1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1	prevA	(prevA)	1 0 1 1
2	OpA + 6	IEW	1 0 0 0	disp + Rs	(disp + Rs)	0 S 1 1	Rs - S	(Rs - S)	1 S 1 1
3				OpA + 6	IEW	1 0 0 0	OpA + 4	IEW	1 0 0 0

Syntax **UNLINK**

Execution
(FP) → (SP)
((SP)) → (FP)

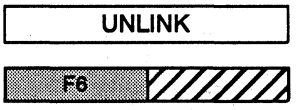
Modes Supported *Operand not necessary for UNLINK*

Status Bits
Z unchanged
N unchanged
C unchanged
V unchanged

Description Unlink and deallocate the current system stack frame:

1) Load the SP (R13) with the contents of the FP (R0).
2) Reload the FP with its previous value (from the system stack).

Instruction Execution Detail



Cy	Address	Data	$\overline{w}$ b $\overline{f}$ $\overline{d}$
1	prevA	(prevA)	1 0 1 1
2	FP	old FP	1 0 1 1
3	OpA + 4	IEW	1 0 0 0

Syntax **XNOR{B}**

Execution NOT (source XOR destination) → destination

Mode Supported *Rs,Rd*

Status Bits

Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C unchanged
V cleared

Description Logically exclusive OR the contents of the source register with the contents of the destination register and return the ones complement of the result.

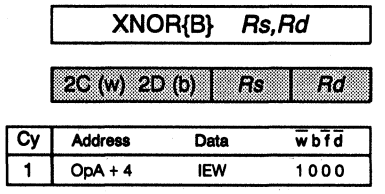
For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will always be FFh. Note that when *Rs* is *ZR*, the instruction is equivalent to NOT *Rd*.

Status bits are set with respect to the size (byte or word) of the operation.

Example

```
Label   XNOR   R2,R11    ; Exclusive OR the values in
                         ;  R2 with R11. Store results
                         ;  in R11.
```

Instruction Execution Detail



Cy	Address	Data	$\overline{w}\,b\,\overline{f}\,\overline{d}$
1	OpA + 4	IEW	1000

Syntax **XOR{B}**

Execution (source) XOR (destination) → (destination)

Modes Supported
Rs,Rd
*Rs,*disp16[Rd]*
#imm16,Rd
*#imm16,*disp16[Rd]*

Status Bits
Z set if the result is zero; cleared otherwise
N equals the most significant bit of the result
C unchanged
V cleared

Description Logically exclusive OR the source operand contents with the contents of the destination operand. Place results in the destination.

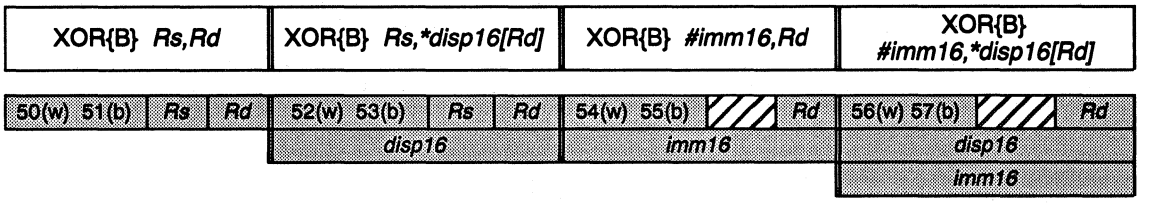
For byte operations, the byte operands are zero-extended to words, are operated on as words, and produce a word result. The most significant byte of the result will always be 00h. Nonregister destinations receive the least significant byte of the result, while registers receive the entire word.

Status bits are set with respect to the size (byte or word) of the operation.

Example

```
LABEL    XORB    #10110011b,R2   ; Exclusive OR the LS
                                 ;  byte of R2 with the
                                 ;  source binary value.
                                 ;  Place results in R2
                                 ;  with the MS byte all
                                 ;  zeroes.
```

Instruction Execution Detail



Cy	Address	Data	w b i d	Address	Data	w b i d	Address	Data	w b i d	Address	Data	w b i d
1	OpA + 4	IEW	1 0 0 0	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1	OpA + 4	IW	1 0 0 1
2				disp + Rd	(disp + Rd)	1 S 1 1	OpA + 6	IEW	1 0 0 0	disp + Rd	(disp + Rd)	1 S 1 1
3				prevA	(prevA)	1 0 1 1				prevA	(prevA)	1 0 1 1
4				disp + Rd	result	0 S 1 1				disp + Rd	result	0 S 1 1
5				OpA + 6	IEW	1 0 0 0				OpA + 6	IEW	1 0 0 0

Appendix A

Glossary

This appendix provides definitions of terms and concepts unique to cMCU™ devices. Other common terms are included if the use of those terms varies from generally accepted usage.

A

absolute address: An addressing mode in which code or operands produce the actual address.

A/D pins: The 18 pins that connect the A/D module to the external world; includes analog inputs AN0–15 and the high and low reference voltages, V_{refhi}, and V_{reflo}.

addressing mode: The method by which an instruction calculates the location of its required data.

AN0–AN15 pins: The 16 analog input channels to the A/D converter's digital inputs.

analog-to-digital (A/D) converter: The cMCU370 A/D Converter, which receives analog data from up to 16 multiplexed inputs.

assembly language: A symbolic language that describes the binary machine code in a more readable form and that can be read by an assembler for conversion into machine code.

asynchronous communications mode: A serial communications format that needs no synchronizing clock. This format begins with a start bit, is followed by data bits and an optional parity bit, and ends with one or two stop bits. This format is commonly used with RS-232-C communications and PC serial ports.

B

BCD: *Binary coded decimal.* Each 4-bit nibble expresses a digit from 0–9 and usually packs two digits to a byte, giving a range of 0–99.

baud: The communication speed for serial ports; equivalent to one bit per second.

C

code address: A value that, when placed in the program counter, is placed on the 16 most significant address lines with the least significant address line set to 0. This effectively multiplies the code value by 2 and makes it possible to address memory of up to 128K bytes.

constant: A value that does not change during execution.

CPU: *Central processing unit.* The cMCU370 product's CPU is register-oriented with a status register, program counter register, and stack pointer. The CPU uses the register file, accessed in one bus cycle, as working registers. The cMCU370 CPUs are the TMS370C8 (8 bit) and the TMS370C16 (16 bit).

D

device: The entire microcontroller, consisting of the CPU and the selected modules integrated on a single chip.

E

edge detection: A process that senses an active pulse transition on a given timer input and provides appropriate output. The active transition can be configured to be low-to-high or high-to-low.

EEPROM: *Electrically erasable programmable read only memory.* Memory that can be programmed and erased under direct program control.

F

freeze bit: A bit "frozen" to an unmodifiable 1 or 0 value, according to customer requirements, during manufacturing.

H

halt mode: A mode that reduces operating power by stopping the internal clock, which stops processing in all the modules. This is the lowest-power mode in which all register contents are preserved.

I

Idle mode: A mode in which the CPU stops processing and waits for the next interrupt. This is not a low-power mode.

immediate operand: An operand whose actual constant value is specified in the instruction.

instruction: The basic unit of programming that causes the execution of one operation; consists of an opcode and operands along with optional labels and comments.

INT1, INT2, and INT3 pins: Pins connected to external devices to allow them to interrupt the CPU. INT1 and INT2 can be software configured as non-maskable interrupts.

interrupt: A signal input to the CPU to stop the flow of a program and force the CPU to execute instructions at an address corresponding to the source of the interrupt. When the interrupt is finished, the CPU resumes execution at the point where it was interrupted.

isosynchronous communications mode: An SCI mode in which data transmission is synchronized by a clock signal (SCICLK) common to both the sender and receiver. The format is identical to the asynchronous mode and consists of a start bit, data bits, an optional parity bit, and a stop bit.

M

machine code: The actual binary values read by the CPU for instruction execution; usually organized as hexadecimal bytes in an assembler listing.

memory map: A map of the address space accessed by the TMS370C16 processor, partitioned according to functionality (memory, registers, etc.).

mnemonic: An alphanumeric symbol designed to aid human memory; commonly represents the opcode of an assembly language instruction.

module: An element that provides a specific function such as a serial interface, memory area, A/D conversion, etc. Such modules are integrated with the CPU to form a device for a specific application.

multiprocessor communications: An SCI format option that enables one processor to efficiently send blocks of data to other processors on the same serial link.

N

nested interrupt: An interrupt that suspends the service routine of a prior interrupt. An executing interrupt can set the ST register's interrupt mask to prevent being suspended by another interrupt.

NMI: *Nonmaskable interrupt.* An interrupt that causes a context switch, once the present instruction finishes execution. When executing, the NMI cannot be interrupted by other NMIs or peripheral interrupts unless an RTI instruction occurs or an ST interrupt bit, L2 – L0, is cleared.

O

offset: A signed value that is added to the base operand to give the final address.

opcode: *Operation code.* In most cases, the first byte of the machine code that describes to the CPU the type of operation and combination of operands. TMS370C16 instructions use 16-bit opcodes.

operand: The part of an instruction designating where the CPU will fetch or store data.

P

prescaler: A circuit that slows the rate of a clocking source to the counter.

prototyping device: A device used before a masked ROM device is available that has identical functions, pinout, size, and timings to the ROM device. Programmable memory such as EEPROM or EPROM is used in place of the masked ROM.

PWM: *Pulse width modulation.* A serial signal in which the information is contained in the width of a pulse of a constant frequency signal. A cMCU370 device can output a PWM signal with a constant duty cycle without any program intervention by using the timer compare features.

R

ratiometric conversion: An analog-to-digital conversion in which the conversion value is a ratio of the V_{ref} source to the analog input. As V_{ref} is increased, the input voltage needed to give a certain conversion value changes, but all conversion values keep the same relationship to V_{ref}.

register file (RF): The 16-register file residing in the CPU. Several registers also serve, respectively, as the frame pointer (R0), implied register (R1), stack pointer (R13), status register (R14), and zero register (R15). Each register is 16 bits.

$\overline{\text{RESET}}$ pin: A pin that when held low starts hardware initialization and ensures an orderly software startup.

S

serial communications interface (SCI): An optional PRISM library module that provides a serial interface, programmable to be asynchronous or isosynchronous. Many timing, data format, and protocol factors are programmable and controlled by the SCI module in operation.

SCICLK pin: *Serial communications interface clock pin.* A pin used as a synchronizing clock input or output in the isosynchronous mode, or as a general-purpose I/O pin.

serial peripheral interface (SPI): An optional PRISM library module that provides a serial interface to facilitate communication between networked master and slave CPUs. As in the SCI, the SPI is set up by software; from then on, the CPU takes no part in timing, data format, or protocol.

signed integer: A number system used to express positive and negative integers.

SPI: See serial peripheral interface.

stack: A designated part of memory used as a last-in, first-out memory for temporary variable storage; used during interrupts and calls to store the current program status. The area occupied by the stack is determined by the stack pointer and the application program.

stack pointer (SP): A CPU register that points to the last entry or top of the stack. The SP is automatically incremented before data is pushed onto the stack and decremented after data is popped (pulled) from the stack.

standby mode: A power reduction mode in which the CPU stops processing, but the on-chip oscillator remains active. Timers remain active and can cause the CPU to exit the standby mode.

status register (ST): A CPU register that monitors the operation of the instructions and contains the global interrupt enable mask bits.

T

TBA (trap table base address): The beginning address of the trap vectors. An algorithm value involving the trap enumeration value is added to this address to find the vector corresponding to the trap. See the TRAP instruction description in Chapter 5.

TRAP: A trap-to-subroutine assembly language instruction that is a subroutine call. Its operand is a trap number that identifies a location in the trap vector table, which contains the address of the subroutine.

U

unsigned integer: A number system used to express positive integers.

W

watchdog timer: A timer option that can be programmed to generate an interrupt when it times out. This provides a hardware monitor over the software to prevent a "lost" program.

Index

C

D

E

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Printed in U.S.A., March 1994

SPNU043