Customer Order No. NSP-INST-REF-M
Publication Number 420010099-001B
June 1984

Series 32000™

Series 32000
Instruction Set Reference Manual

2900 Semiconductor Drive
Santa Clara, California 95051

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	08/83	First Release. NS16000 Instruction Set Reference Manual. Publication No. 420010099-001
B	06/84	This manual is being reissued to reflect the Series 32000 name change. There are also minor technical changes.

PREFACE

The Series 32000™ family is the latest entry into the high performance microprocessor marketplace. The family is carefully designed and optimized to operate in environments which demand large-scale computing capabilities, but require only microprocessor size and price. A great deal of concern has been given to every aspect of the architecture to ensure high performance while not compromising other considerations, such as memory management or code density.

National Semiconductor takes great pride in introducing the Series 32000 family of microprocessors and peripheral components, which offers a total system solution to a wide variety of new microprocessor applications.

The following National Semiconductor publications provide related study/reference material for using the Series 32000 family.

Series 32000 NSX Cross-Support Utilities Reference Manual	(Pub. No. 420306617-002)
Series 32000 Pascal Language and Compiler Reference Manual	(Pub. No. 420306618-002)
Series 32000 Cross-Assembler Reference Manual	(Pub. No. 420306619-002)
Series 32000 ISE16: NS32016 and NS32008 In-System Emulators User's Manual	(Pub. No. 420306675-002)
Series 32000 Symbolic Debugger Reference Manual	(Pub. No. 420306676-002)
Series 32000 Run-Time Support Library Reference Manual	(Pub. No. 420308038-002)
Series 32000 Floating-Point Support Library Reference Manual	(Pub. No. 420308220-002)
Series 32000 Development Board Monitor Reference Manual	(Pub. No. 420308221-002)
Series 32000 NSX Operations Manual	(Pub. No. 424009011-002)
Series 32000 DB32000 Development Board User's Manual	(Pub. No. 420010144-001)
Series 32000 TDS: Tiny Development System	(Pub. No. 420306440-001)
Series 32000 DB32016 Development Board User's Manual	(Pub. No. 420310111-001)
Series 32000 EXEC: ROMable Real-Time Multitasking EXECUTIVE Reference Manual	(Pub. No. 420010206-001)
Series 32000 GENIX Cross-Support Software Programmer's Manual	
Volume 1	(Pub. No. 424010106-001)
Volume 2	(Pub. No. 424010106-002)
Series 32000 GENIX Programmer's Manual	
Volume 1	(Pub. No. 424308225-001)
Volume 2	(Pub. No. 424308225-002)
Series 32000 GENIX Debugging Reference Manuals	
GENIX ISE16: NS32008 and NS32016 In-System Emulators	(Pub. No. 420308165-001)
GENIX Symbolic Debugger	(Pub. No. 424010149-001)
Series 32000 SYS32 System Manual	(Pub. No. 420308225-001)

CONTENTS

CONTENTS (Cont.)

ILLUSTRATIONS

TABLES

Chapter 1

INTRODUCTION

This document is a revised definition of the Series 32000 instruction set. It provides more specific information on architectural details, and also incorporates further information on compatibility issues.

This is not a full architectural description, and is intended to supplement and update other documentation already in print. Specific areas not included here are:

- Material which is primarily tutorial in nature.
- Details of memory management. See instead the NS32082 MMU data sheet.

The term "undefined" is used frequently as the outcome of an illegal instruction form. An outcome which is architecturally undefined is not guaranteed to remain the same under all conditions, in all component revisions, or in future expanded implementations of this architecture. Many of these illegal options may "work" in the current implementation, but they are nevertheless considered undefined by NSC, and should always be avoided. Illegal instruction forms, when executed in User mode, are guaranteed not to bypass any of the protection mechanisms implemented in the Series 32000 family.

The manual is divided as follows:

1. INTRODUCTION

2. PROGRAMMING MODEL
   Definitions of the Series 32000 register set and other resources visible to the programmer.

3. INSTRUCTIONS AND DATA TYPES
   A discussion of the instruction set by functional groups, including definitions of associated data types and exceptional conditions.

4. INSTRUCTION OPTIONS AND CONSTRUCTION
   Definitions of the Series 32000 addressing modes and the construction of instructions in assembly language and binary.

5. INSTRUCTION SET
   Individual definitions of the Series 32000 instructions, organized alphabetically by mnemonic.

6. EXCEPTION PROCESSING
   Definitions of the Series 32000 interrupt and trap structure, including the response to a Reset.

Appendices:

A. LIST OF INSTRUCTIONS BY FUNCTIONAL GROUP
B. INSTRUCTION EXECUTION TIMING

Chapter 2

PROGRAMMING MODEL

This chapter defines the programming model (resources visible to the programmer) presented by the Series 32000 architecture. More specifically, this chapter presents the Series 32000 register set, memory organization, and the functions of dedicated memory areas used by Series 32000 hardware. Also presented here is the mechanism used to protect privileged portions of the programming model.

This chapter is organized as follows:

2.1 General Registers

There are eight 32-bit General-Purpose registers, named R0 through R7 (see Figure 2-1). The contents of any General-Purpose register can be used as:

1. Data, using the Register addressing modes (Section 4.4.1).

2. A base pointer, using the Register Relative addressing modes (Section 4.4.2).

3. An index value, using the Scaled Indexing modifier in an addressing mode (Section 4.4.9).

Data held within a General-Purpose register may be treated as an 8-bit, 16-bit, or 32-bit value. When an instruction operates on data of less than 32 bits, the value used is the low-order portion of the register. The remaining portion of the register is neither used nor affected.

For extended arithmetic (the MEIi and DEIi instructions), the General-Purpose registers are combined to form even/odd register pairs: R0/R1, R2/R3, R4/R5, and R6/R7. See Section 4.4.1 for details of this use.

2.2 Dedicated Registers

The Dedicated registers store memory addresses and general status information (see Figure 2-1). The eight Dedicated registers are:

- Program Counter (PC)
- Static Base Register (SB)
- User Stack Pointer (SP1)
- Interrupt Stack Pointer (SP0)
- Frame Pointer (FP)
- Interrupt Base Register (INTBASE)
- Module Register (MOD)
- Processor Status Register (PSR)

The PC, SB, SP1, SP0, FP, and INTBASE registers each hold 32-bit memory addresses. The MOD and PSR registers are each 16 bits long. The MOD register contains a memory address, and the PSR register contains status information. The addresses contained in these registers are interpreted as virtual in memory-managed systems.

Because the current implementation of the Series 32000 family uses only 24-bit addresses, only the low-order 24 bits of the 32-bit registers are implemented. The high-order eight bits are permanently zero for reasons of upward compatibility.

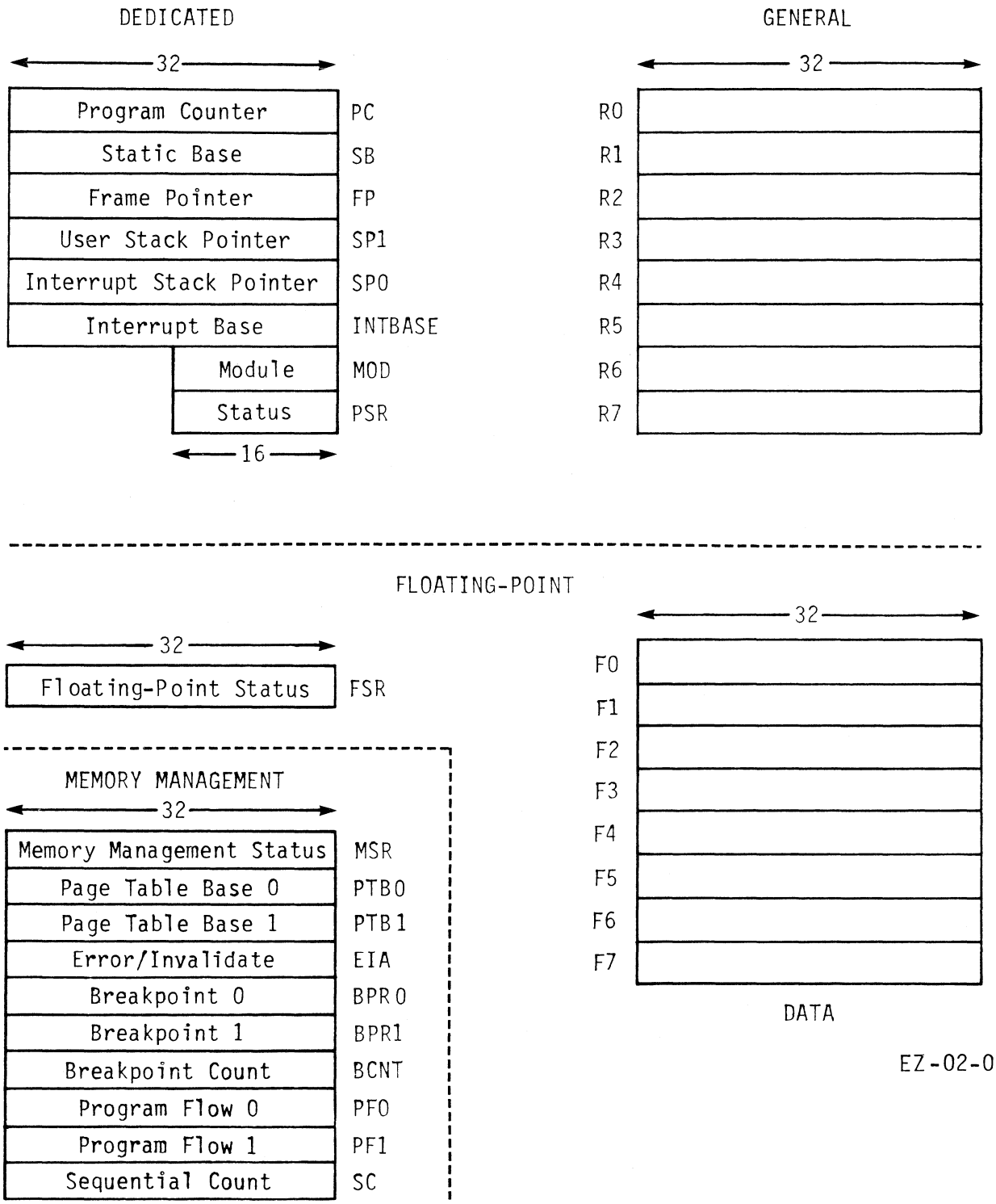


Figure 2-1 Series 32000 Register Set

A description of each Dedicated register follows.

PC — The Program Counter is available as a Base register (using the Program Memory addressing mode, Section 4.4.8). It contains the memory address of the first byte of the instruction currently being executed. The PC is incremented (to point to the next instruction) only when the current instruction is completed. On occurrence of a Reset (Chapter 6), the PC is set to zero, and the first instruction is fetched from this address.

SP1 — The User Stack Pointer points to the top of the User Stack (Section 2.7.1). The SP1 register is selected for all stack operations while the S bit in the Processor Status Register is set to 1.

SP0 — The Interrupt Stack Pointer points to the top of the Interrupt Stack (Section 2.7.1). The Interrupt Stack is selected for all stack operations while the S bit in the PSR is set to 0. It is also automatically selected whenever an interrupt or trap occurs. In memory-managed systems, SP0 must always contain a valid Supervisor-Mode virtual address (see Section 2.7.1).

NOTE: The SP1 and SP0 registers are never referenced directly by a program. Instead, the symbol "SP" is used, meaning the Stack Pointer which is currently selected. This SP register is available as a base pointer using the Stack Memory and Stack Memory Relative addressing modes (Sections 4.4.8 and 4.4.3). The Top of Stack addressing mode uses the SP register in performing "push" and "pop" references to the top of the stack (Section 4.4.7).

FP — The Frame Pointer points to a dynamically-allocated data area created at the beginning of a procedure (by the ENTER instruction). This area is generally called the "activation record" for the procedure, and contains its parameters, local variables, saved registers, and return address. The FP register is available as a base pointer using the Frame Memory and Frame Memory Relative addressing modes (Sections 4.4.8 and 4.4.3).

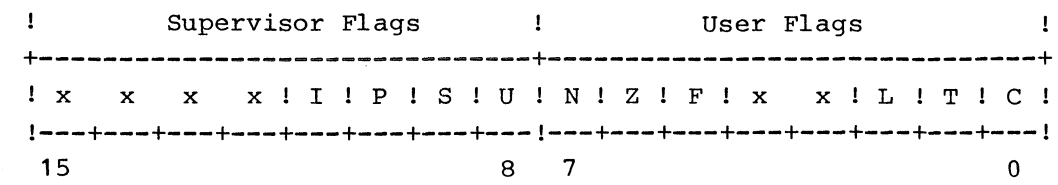
INTBASE — The Interrupt Base register contains the base address of the Interrupt Dispatch Table. This is a vector table which contains the descriptors of the trap and interrupt service procedures. See Chapter 6 for details of trap and interrupt handling. In memory-managed systems, INTBASE must always contain a valid Supervisor-Mode virtual address (see Section 2.7.4).

MOD The Module register points to the current module's Module Table entry. The Module Table entry is a 16-byte block of memory containing three pointers for the current module:

- SB (Static Base, a pointer to its static data area)
- LB (Link Base, a pointer to its Link Table)
- PB (Program Base, a pointer to the beginning of its code)

See Section 2.7.2.

SB The Static Base register contains the base address of data which has been statically allocated (i.e. allocated once, before program execution) to the current module. This address is a copy of the SB pointer in the current Module Table entry. It is available for use in the Static Memory and Static Memory Relative addressing modes (Sections 4.4.8 and 4.4.3). The Static Base register is automatically updated whenever control is transferred from one module to another.



x = reserved

Figure 2-2 Processor Status Register

PSR, UPSR

The Processor Status Register (Figure 2-2) contains 16 mode and status flag bits, of which 10 bits are currently implemented. All implemented PSR flags are readable and writable. The bit positions marked "x" in Figure 2-2 are reserved for future use. They are not currently implemented, and do not retain information written to them. For upward compatibility reasons, no program should attempt to change these bits, nor should any program assume that they are always zero (even though they appear to be permanently zero in the current implementation).

The least-significant byte of the PSR contains flags which are always accessible. This byte is also called the UPSR, for "User PSR".

The most-significant byte of the PSR contains the Supervisor flags. Supervisor flags are accessible only by a program running in Supervisor mode (see the discussion of the U bit which follows). Any attempt by a User Mode program to load, store or modify this byte causes the Illegal Operation trap, Trap (ILL), instead. See Section 2.8 for further details of protection features.

Upon occurrence of an interrupt or trap, the PSR is pushed onto the Interrupt Stack. Certain PSR bits are then automatically cleared (as stated in their descriptions) to establish the proper modes of operation for interrupt service. See Chapter 6 for further details of interrupt and trap service.

NOTE: The PSR P bit is sometimes cleared before the PSR is pushed onto the Interrupt Stack. See Chapter 6.

All implemented PSR flags are cleared to zero on occurrence of a Reset (Chapter 6).

User PSR Flags

C is the Carry flag. The Carry flag signals a carry condition during execution of an addition instruction or a borrow condition during a subtraction instruction. If a carry or borrow has occurred, the C bit is set to 1. If no carry or borrow has occurred, the C bit is set to 0. See Section 3.1 for definitions of carry and borrow conditions.

T is the Trace flag. This flag places a program in Trace mode, allowing step-by-step inspection of the effects of each instruction. While the T bit is set, the Trace trap, Trap (TRC), occurs at the completion of each instruction. The T bit interacts with the P bit to ensure correct operation of Trace Mode regardless of any interrupts or other traps which may also be occurring. It is cleared on occurrence of any trap or interrupt. See Chapter 6 for further details of this trap and of trap service.

L is the Low flag. The Low flag signals the result of an unsigned comparison between two integers. (All integer comparison instructions perform both signed and unsigned comparisons.) If the second operand of a comparison instruction is less than the first, the L bit is set to 1. If the second operand is greater than or equal to the first, the L bit is set to 0. The L flag is always cleared by the floating-point comparison instruction (CMPf).

F is the F Flag. The F flag is a general condition flag, used by various instructions to signal exceptional conditions (e.g. integer overflow from addition or subtraction), or to distinguish among outcomes (e.g. what condition has caused a String instruction to terminate).

Z is the Zero flag. The Zero flag indicates the result of comparing two integers or two floating-point values. If they are equal, the Z bit is set to 1. If they are not equal, the Z bit is set to 0.

N is the Negative flag. The Negative flag indicates the result of a signed comparison between two integers or two floating-point values.

NOTE: The integer comparison instructions, CMPi and CMPQi, perform both signed and unsigned comparisons.

If the second operand is less than the first, the N bit is set to 1. If the second operand is greater than or equal to the first, the N bit is set to 0.

The N, Z, F, L and C bits constitute a "condition" which may be used by the Conditional Branch (Bcond) and Save Condition Code (Scondi) instructions. In addition, the F bit may be used to cause a trap (by the FLAG instruction).

Supervisor PSR Flags

U is the User Mode flag. If the U bit is 1, the current program is running in User mode, and may not use privileged instructions or reference protected registers. If the U bit is 0, the current program is running in Supervisor mode, and is not restricted. In memory-managed systems, address translation and memory protection features may also be affected by the state of this bit. The U bit is automatically cleared on occurrence of any interrupt or trap. See Section 2.8 for further details of protection features.

S is the Stack flag. The S bit selects which of the two stack pointers is to be used for stack operations. If the S bit is 1, the User Stack Pointer (SP1) is selected. If the S bit is 0, the Interrupt Stack Pointer (SP0) is selected. The S bit is automatically cleared on occurrence of a trap or interrupt.

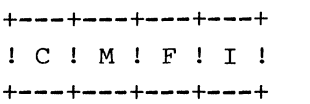
P is the Trace Trap Pending flag. The P bit interacts with the T bit to ensure correct trace results in programs which are being interrupted or trapped. It is automatically cleared on occurrence of any trap or interrupt. The P bit in the PSR image which is pushed on occurrence of an interrupt or trap may also be cleared, depending on the trap or interrupt. See Chapter 6 for further details of Trace mode.

I is the Interrupt Enable flag. If the I bit is 1, both Maskable and Non-Maskable interrupts are accepted. If the I bit is 0, only Non-Maskable interrupts are accepted. The I bit is automatically cleared on occurrence of an interrupt or the Abort trap, Trap (ABT). No other traps affect this bit, and this bit does not disable traps when clear. Interrupts are described in Chapter 6.

2.3 Configuration Register (CFG)

The Configuration register is used to enable or disable certain Series 32000 features which are currently optional. The only operation performed on this register is to load it using the SETCFG instruction, which is intended to be executed only after a Reset to declare the system's configuration.

The CFG register is four bits in length and has the form



where the bits correspond to features as given below.

I Interrupt vectoring. This bit declares whether hardware support is available for direct vectoring of maskable interrupts. If the I bit is set, service of a maskable interrupt includes reading an 8-bit value which selects an Interrupt Dispatch Table entry to use in locating the interrupt service procedure (see Section 2.7.4). This 8-bit value is supplied by an NS32202 Interrupt Control Unit. If the I bit is not set, maskable interrupts are not vectored, and use by default the first entry (NVI) of the Interrupt Dispatch Table, requiring no hardware support.

F Floating-Point instruction set. If this bit is set, the Floating-Point instruction set (Section 3.3) is enabled, and an attached NS32081 Floating-Point Unit will be used to execute these instructions. If the F bit is not set, all Floating-Point instructions generate Trap (UND) instead. (The trap mechanism employed by the Series 32000 architecture allows software to intercept this trap and fully emulate the functions of the NS32081.)

M Memory Management instruction set. If this bit is set, the LMR, SMR, RDVAL and WRVAL instructions (Section 3.12) are enabled, and an attached NS32082 Memory Management Unit will be used to execute them. If the M bit is not set, these instructions generate Trap (UND) instead. (Note: the Memory Management instructions MOVSUi and MOVUSi are not affected by this bit, and are always available.)

C Custom instruction set. If this bit is set, the Custom instruction set (Section 3.13) is enabled, and will use attached custom hardware (unique to a given system). If it is not set, all Custom instructions generate Trap (UND) instead.

2.4 Floating-Point Registers

Floating-Point registers are present in systems supporting the Floating-Point instruction set (either by using the NS32081 Floating-Point Unit or by software emulation). See Figure 2-1. There are eight Floating-Point Data registers (F0-F7) and one Floating-Point Status register (FSR).

2.4.1 Floating-Point Data Registers

The Floating-Point Data registers provide a high-speed workspace for floating-point operations. These registers are named F0 through F7, and are 32 bits in length. They are referenced whenever the Register addressing mode (Section 4.4.1) is used in a floating-point instruction to specify the location of a floating-point operand. Floating-point operands are located in memory or in Floating-Point Data registers, and integer operands are located in memory or in General-Purpose registers.

The Floating-Point Data registers may be used individually to hold 32-bit single-precision floating-point numbers, or they may be used in even/odd pairs (F0/F1, F2/F3, F4/F5, F6/F7) to hold 64-bit double-precision floating-point numbers. When a double-precision operand is held in a register pair, the even register holds the low-order half of the number, and the odd register holds the high-order half. A register pair is specified using the name of its even register.

2.4.2 Floating-Point Status Register (FSR)

The Floating-Point Status register (FSR) selects operating modes and records any exceptional conditions encountered during execution of a floating-point instruction. Figure 2-3 shows the format of the FSR.

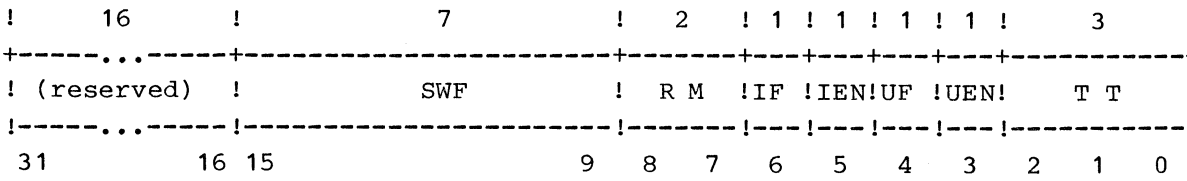


Figure 2-3 Floating-Point Status Register

Bits 9 through 31 of the FSR are reserved. The SWF field (bits 9 through 15) is currently reserved for NSC software use (floating-point extension software). Information written to this field is retained, but does not affect any hardware operations. The remaining bits (16 through 31) are not implemented, and do not retain information written to them. For upward compatibility reasons, no program should attempt to change either reserved field, nor should any program assume that their contents are always zero (even though bits 16-31 appear to be permanently zero in the current implementation). To change the contents of the FSR, the following procedure should always be followed:

1. Use the SFSR instruction to store the FSR in a temporary location.

2. Change the desired fields in this temporary copy.

3. Use the LFSR instruction to load the temporary copy into the FSR.

FSR Mode Fields

The FSR mode fields are set by the programmer to establish modes of operation for floating-point instructions. The mode fields are encoded as follows.

RM Rounding Mode: bits 7 and 8. This field selects the rounding method to be used whenever a floating-point result cannot be exactly represented in the format of the destination operand. The rounding modes are:

00 Round to nearest value. The value which is nearest to the exact result is selected. If the result is exactly halfway between the two nearest values, the even value (LSB = 0) is delivered to the destination.

01 Round toward zero. The nearest value whose absolute value is less than, or equal to, the exact result is delivered to the destination.

10 Round toward positive infinity. The nearest value which is greater than, or equal to, the exact result is delivered to the destination.

11 Round toward negative infinity. The nearest value which is less than, or equal to, the exact result is delivered to the destination.

UEN Underflow Trap Enable: bit 3. If this bit is set, Trap (FPU) occurs whenever an underflow condition is encountered. See Section 3.3.7 for the definition of floating-point underflow. If it is not set, any underflow condition returns a result of positive zero (Section 3.3.3), and no trap occurs.

IEN Inexact Result Trap Enable: bit 5. If this bit is set, Trap (FPU) occurs whenever the result of a floating-point instruction is not exact. If it is not set, the result is rounded according to the selected rounding mode, and no trap occurs.

FSR Status Fields

The FSR status fields record exceptional conditions encountered during the execution of a floating-point instruction. The meanings of the FSR status bits are as follows:

TT Trap Type: bits 0-2. This 3-bit field records any exceptional condition detected by a floating-point instruction. These conditions are defined in Section 3.3.7. They are reported as:

000	No exceptional condition occurred.
001	Underflow
010	Overflow
011	Division by Zero
100	Illegal Instruction
101	Invalid Operation
110	Inexact Result
111	(Reserved for future use.)

The TT field is loaded with zero whenever any floating-point instruction except LFSR or SFSR completes without encountering an exceptional condition. It is also set to zero by a Reset (Chapter 6) or by writing zero into it with the Load FSR (LFSR) instruction. Underflow and Inexact Result are always reported in the TT field, regardless of the settings of the UEN and IEN bits.

UF Underflow Flag: bit 4. This bit is set whenever an underflow condition is detected. See Section 3.3.7 for the definition of floating-point underflow. The function of the UF bit is not affected by the state of the UEN bit. The UF bit is cleared only by writing a zero into it with the LFSR instruction or by a Reset (Chapter 6).

IF Inexact Result Flag: Bit 6. This bit is set whenever an Inexact Result condition is detected, and no other errors have occurred. See Section 3.3.7 for the definition of this condition. It is cleared only by writing a zero into it with the LFSR instruction or by a Reset (Chapter 6).

2.5 Memory Management Registers

Memory Management registers are present in systems incorporating the Series 32000 memory management option. These registers are currently implemented in the NS32082 Memory Management Unit and are made available by setting the M bit in the CFG register (Section 2.3). There are ten 32-bit Memory Management registers (Figure 2-1):

MSR	Memory Management Status Register
PTB0, PTB1	Page Table Base Registers
EIA	Error/Invalidate Address Register
BPR0, BPR1	Breakpoint Registers
BCNT	Breakpoint Count Register
PF0, PF1	Program Flow Registers
SC	Sequential Count Register

The four registers MSR, PTB0, PTB1 and EIA provide status and control functions for implementing memory management (mapping and protection) functions.

The three registers BPR0, BPR1 and BCNT implement hardware breakpointing on read, write and/or execute accesses to specified addresses.

The three registers PF0, PF1 and SC provide the capability of tracing the flow of a program backward from the point of a breakpoint or error.

The Memory Management registers are each 32 bits in length. The following describes briefly the function of each register.

MSR — contains the memory management status and control flags.

PTB0 and PTB1 — support virtual memory and address translation. These registers contain the base addresses of the Level 1 Page Tables.

EIA — supports virtual memory and address translation. When read, this register contains the virtual address that caused the most recent translation error. When written to, this register causes the removal of an invalid Page Table entry from the Translation Buffer.

BPR0 and BPR1 — support program debugging. These registers contain the breakpoint addresses and the breakpoint conditions. The processor breaks program execution when these addresses and conditions are met.

BCNT — supports program debugging. This register allows a breakpoint address to be ignored for a specified number of times.

PF0 and PF1 — support program debugging. These registers contain the addresses of the two most recent nonsequentially fetched instructions.

SC — supports program debugging. This register contains two 16-bit fields:

SC1, containing the number of sequential instructions executed between the last two nonsequentially fetched instructions, and

SC0, containing the number of sequential instructions executed between the last nonsequentially fetched instruction and the point where program flow tracing was terminated.

2.6 Memory Organization

The Series 32000 architecture supports a memory addressing space of four gigabytes (corresponding to a 32-bit address) of which the lower 16 megabytes (24-bit address) is currently implemented.

2.6.1 Addressing

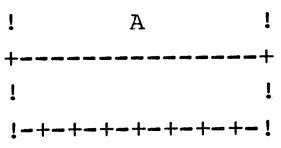
A memory address is a 32-bit unsigned integer. It uniquely identifies an 8-bit location (a byte) within the memory space. In the current implementation only the least-significant 24 bits of an address are used, interpreted as a 24-bit unsigned number. In decimal, the resulting 24-bit addressing range is 0 through 16,777,215.

NOTES: 1. Addresses outside the above range, including negative addresses, are undefined because their interpretations will differ between systems implementing different maximum addressing spaces. Do not make use of "wrap-around" features of any implementation for generating addresses.

2. Except where otherwise indicated, all addresses and memory spaces given in this manual are virtual in memory-managed systems, and can be mapped to any "physical" (or "real") memory page. In the current implementation, a separate 16-Mbyte memory space can be made available to each user program, regardless of the size of physical memory.

2.6.2 Memory Operand Formats

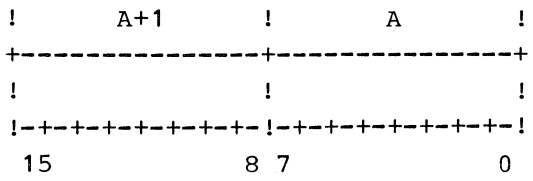
The basic storage unit is the byte. A byte holds eight bits of data and has the following form:



Byte at Address A

Bit positions are numbered from 0 to 7. Bit 0 is the least-significant bit; bit 7 is the most-significant bit.

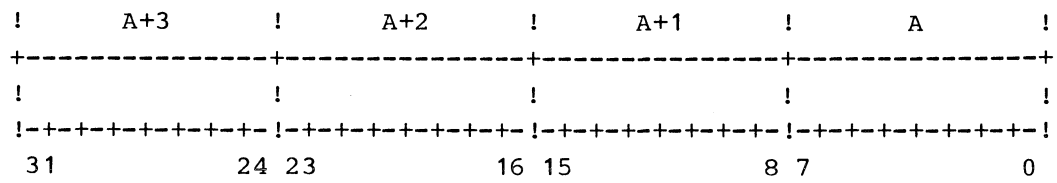
A 16-bit value is called a word. It is held in memory as a pair of contiguous bytes.



Word at Address A

The byte at the lower address is the least-significant byte; the byte at the higher address is the most-significant byte. A word has the same address as its least-significant byte and may start at any address.

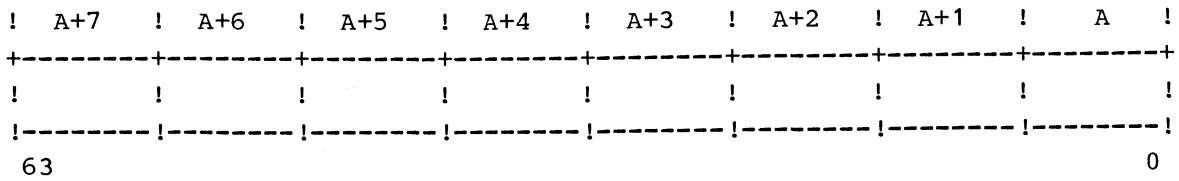
A 32-bit value is called a double-word. It is held in memory as four contiguous bytes. A double-word can hold either a 32-bit integer or a single-precision floating-point value.



Double-word at Address A

The least-significant byte of a double-word is stored at the lowest address. A double-word has the same address as its least-significant byte and may start at any address.

A 64-bit value is called a quad-word. It is held in memory as eight contiguous bytes. A quad-word can hold a 64-bit integer or a double-precision floating-point value.



Quad-word at Address A

The least-significant byte of a quad-word is stored at the lowest address. A quad-word has the same memory address as its least-significant byte and may start at any address.

2.6.3 Data Alignment

With the sole exception of the Page Tables used for memory management, there are no alignment restrictions in the Series 32000 architecture. Operands of any length may start at any byte address.

For optimal throughput, however, it is usually desirable to align data. A method for alignment which applies well to all memory bus size implementations (8, 16 or 32 bits) is to align operands on "integral" boundaries. By this method, words are stored at even addresses, double-words at multiples of four, and quad-words at multiples of eight.

2.7 Dedicated Memory Areas

A Series 32000-based system will make use of certain designated memory areas for the following purposes:

- User and Interrupt Stacks
- Module Table
- Link Tables
- Interrupt Dispatch Table and Cascade Table
- Input and Output

2.7.1 User and Interrupt Stacks

A stack is a block of memory used as a last-in/first-out (LIFO) buffer. The contents of a Stack Pointer register specify an address within the block, and the value at this address is considered to be at the top of the stack.

There are two stacks: a User Stack and an Interrupt Stack. The User Stack Pointer (SP1) specifies the address of the top of the User Stack, and the Interrupt Stack Pointer (SP0) specifies the address of the top of the Interrupt Stack. At any time, one of these stacks is selected for stack operations (by the PSR S bit, Section 2.2). The User stack is generally assigned to User-Mode programs, although programs running in Supervisor Mode may also select it. The Interrupt stack is identical in function to the User stack, except that it is always selected on a trap or interrupt to receive the return information (return address, MOD and PSR: see Chapter 6). An interrupt or trap service routine may continue to use the Interrupt Stack, or it may re-select the User stack.

Stacks grow downward in memory; i.e., toward lower addresses. To pop a value, the current Stack Pointer is incremented by the value's length in bytes after reading it ("post-increment"). To push a value, the current Stack Pointer is decremented by the value's length in bytes before writing it ("pre-decrement"). In either case, the Stack Pointer indicates the new top of the stack.

Data may be read from, or written to, the currently-selected stack at any time, using the Top of Stack addressing mode (Section 4.4.7), which performs an automatic push or pop, as appropriate. In addition, the current Stack Pointer may be used as a base pointer in the Stack Memory and Stack Memory Relative addressing modes (Sections 4.4.8 and 4.4.3).

The current stack also receives return addresses and other context information saved in the process of invoking a procedure. Examples of this use are the BSR (Branch to Subroutine) instruction and the ENTER (Enter Procedure Context) instruction. Instructions of this type always modify the Stack Pointer in multiples of four, so that the stack may always be kept aligned on 32-bit boundaries if desired for optimal throughput.

NOTES: 1. Information popped from a stack should never be considered still available in its original memory location after the popping instruction terminates, nor should any program ever store information in a memory area which is available for stack expansion but is not within the stack. These requirements are made for reasons of upward compatibility and compatibility between systems.

2. In memory-managed systems, the Interrupt stack must always be available in physical memory. On occurrence of an interrupt or trap, the contents of the Interrupt Stack pointer are treated as a Supervisor-Mode virtual address.

2.7.2 Module Table

The Series 32000 architecture supports software modules and modular programs through a Module Table. This table contains one 16-byte entry (a module descriptor) for each module in the program. The MOD Register (Section 2.2) holds the address of the Module Table entry for the currently-running module.

All Module Table entries need not be held in a single contiguous memory space, but they must all be contained within the first 64K bytes of memory, due to the fact that the MOD register holds only a 16-bit address. A Series 32000-based system, therefore, can hold up to 4096 modules at a time (4096 modules per user, in memory-managed systems).

A module descriptor contains four 32-bit pointers, of which the first three are used in the current implementation. These pointers are found relative to the contents of the MOD register as shown in Figure 2-4.

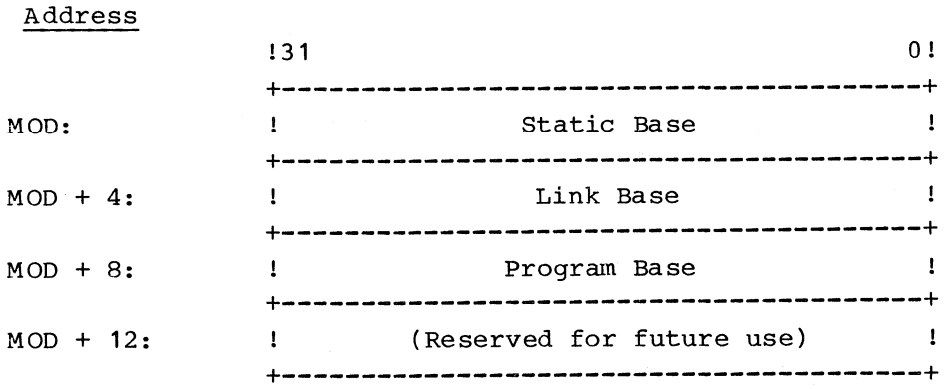


Figure 2-4 Module Descriptor Format

The Static Base pointer contains the address of a memory area allocated to this module for static data; i.e., data which is allocated only once, before execution. This pointer is loaded into the Static Base register whenever control is transferred from one module to another.

The Link Base pointer contains the address of the Link Table assigned to this module. See Section 2.7.3.

The Program Base pointer contains the address of the first byte of the code section of this module. It is used by other modules (through their Link Tables) to transfer control to specific procedures within this module.

NOTES: 1. All Module Table entries must be entirely contained within the first 64K bytes of memory. This means that MOD register values of FFF1 through FFFF (Hex) are reserved.

2. In memory-managed systems, all module descriptors for interrupt or trap service routines must always be in physical memory. The contents of the three pointers are interpreted as Supervisor-Mode virtual addresses.

2.7.3 Link Tables

One Link Table is allocated to each module of a program. The Link Base pointer of the current Module Table entry (Section 2.7.2) points to the Link Table for the currently running module.

Each Link Table provides information which is used for:

1. Sharing variables between modules. Such variables are available to other modules via the External addressing mode (Section 4.4.6).

2. Transferring control from one module to another. This is done directly from the current Link Table via the CXP instruction.

A module's Link Table is constructed by a linker program based on requests made by the module for external items. After allocating all of the modules comprising a program, the linker then fills each Link Table with the information necessary for communication between modules.

The format of a Link Table is given in Figure 2-5. A Link Table entry for an external variable contains the 32-bit address of that variable. An entry for an external procedure contains a 32-bit procedure descriptor consisting of two 16-bit fields: Module and Offset. The Module field holds the new MOD register contents for the module containing the external procedure. The Offset field is an unsigned value giving the position of the external procedure's entry point relative to its module's Program Base pointer (Section 2.7.2).

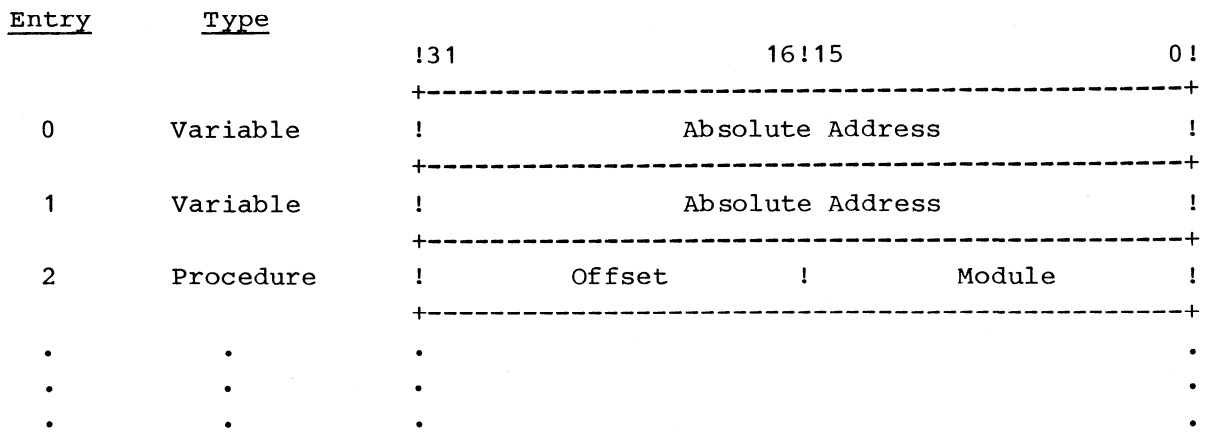


Figure 2-5 Sample Link Table

2.7.4 Interrupt Dispatch Table and Cascade Table

The Series 32000 architecture supports handling of exceptions (traps and interrupts) through the Interrupt Dispatch Table. This table contains procedure descriptors (Section 2.7.3) for locating the service procedures assigned to each exception. The Interrupt Dispatch Table location is given by the INTBASE register.

The Interrupt Dispatch Table contains one 32-bit descriptor for each exception. A Series 32000-based system can process up to 256 exceptions, depending on the system configuration. A Cascade Table may also exist, appended before the Dispatch Table.

For further details of interrupt and trap service, see Chapter 6.

NOTE: In memory-managed systems, the Interrupt Dispatch Table (and Cascade Table, if present) must always reside in physical memory. The INTBASE register contents are interpreted as a Supervisor-Mode virtual address. The Module portion of each procedure descriptor is also interpreted as a Supervisor-Mode virtual address.

2.7.5 Input and Output

Input and output ports are memory-mapped in Series 32000-based systems. That is, all I/O devices are addressed as memory locations, and I/O operations are performed by reading from, or writing to, an I/O device as if it were a byte, word, or double-word of memory. There are no specific input and output instructions.

The hardware design of each individual system defines the number and type of I/O devices as well as the addresses at which they are located. This is not defined by the Series 32000 architecture. However, the current implementation encourages two I/O assignments for interrupt handling, described below.

When a maskable interrupt occurs, an 8-bit vector number is read from address 00FFFE00 (Hex). In memory-managed systems, this is a Supervisor-Mode virtual address, and must always have a valid mapping. Depending on the interrupt configuration mode (Vectored or Non-Vectored, Section 2.3), the vector value may not actually be used, but the read operation always occurs.

When a Non-Maskable Interrupt (NMI) occurs, the processor reads one byte from address 00FFFF00 (Hex). In memory-managed systems, this again is a Supervisor-Mode virtual address, and must always have a valid mapping. The processor does not use the data which was read.

Care should be taken in the system design to ensure that these read operations do not trigger side-effects.

For further details of interrupt service, see Chapter 6 and the applicable CPU data sheet.

2.8 Privilege States and Protection

The Series 32000 family implements two privilege states: User Mode and Supervisor Mode.

The U flag in the PSR determines the privilege state. When the U flag is 1, the system is in User Mode, otherwise it is in Supervisor Mode.

A program running in User Mode is prevented from accessing privileged registers. These registers are:

- The most-significant byte of the Processor Status Register (PSR).
- The INTBASE register.
- The CFG register.
- All Memory Management registers.

The Interrupt Stack Pointer (SP0) is also implicitly protected by the fact that a User-Mode program cannot access the PSR S bit to select it for use.

User-Mode restrictions are enforced by the Illegal Operation trap, Trap (ILL), which occurs whenever a User-Mode program attempts to access a privileged register. Instructions which cause, or may cause, Trap (ILL) are listed in Table 2-1.

Programs running in Supervisor Mode have none of the above restrictions, as they are assumed to be trusted portions of an operating system.

In addition to the above restrictions, memory-managed systems can restrict access to memory pages based on the privilege state. Violations of such access restrictions cause the Abort trap, Trap (ABT). Since I/O devices are mapped as memory, they may also be protected by this mechanism as required.

Table 2-1 Privileged Instructions

Instruction	Mnemonic
Load Processor Register (if INTBASE or PSR)	LPRi
Store Processor Register (if INTBASE or PSR)	SPRi
Bit Clear in PSR (if Word length)	BICPSRW
Bit Set in PSR (if Word length)	BISPSRW
Set Configuration	SETCFG
Return from Trap	RETT
Return from Interrupt	RETI
Load Memory Management Register	LMR
Store Memory Management Register	SMR
Move Value from Supervisor to User Space	MOVSUi
Move Value from User to Supervisor Space	MOVUSi
Validate Address for Reading	RDVAL
Validate Address for Writing	WRVAL

Chapter 3

INSTRUCTIONS AND DATA TYPES

This chapter presents an overview of the Series 32000 instruction set by functional groups and describes the data types and structures on which they act.

The groups by which this chapter is organized are:

Group	Section
Integer Instructions	3.1
Packed Decimal (BCD) Instructions	3.2
Floating-Point Instructions	3.3
Logical Instructions	3.4
Bit Instructions	3.5
Bit Field Instructions	3.6
String Instructions	3.7
Block Instructions	3.8
Array Instructions	3.9
Processor Control Instructions	3.10
Processor Service Instructions	3.11
Memory Management Instructions	3.12
Custom Instructions	3.13

Instructions in each group are listed in three columns.

Instruction: A brief instruction name.

Mnemonic Forms: A list of all forms that the instruction mnemonic may take in assembly language.

Index: The general mnemonic form of the instruction. Chapter 5 (Instruction Set) is organized alphabetically by this index.

3.1 Integer Instructions

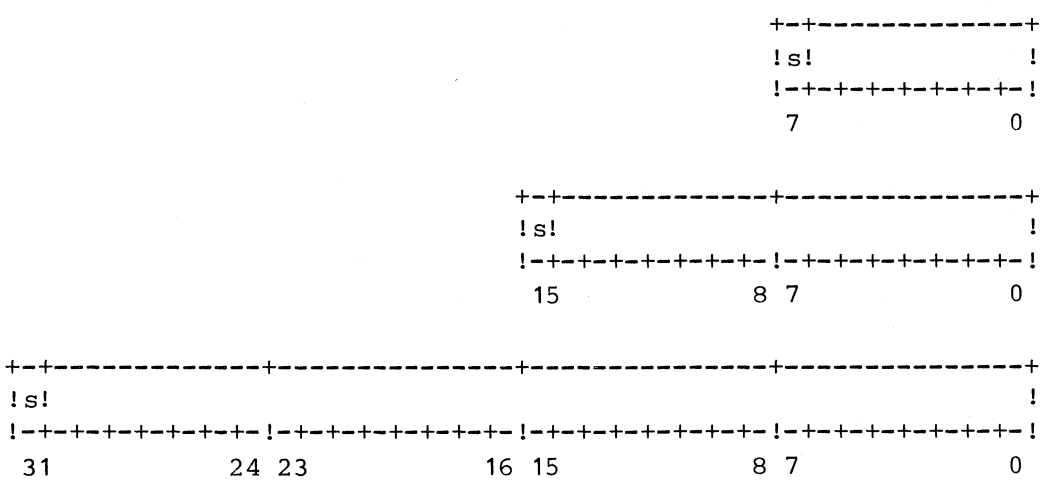
Integer instructions operate on byte, word, and double-word integer operands. The following is a list of the Integer instructions:

Instruction	Mnemonic Forms	Index
Arithmetic		
Add	ADDB, ADDW, ADDD	ADDi
Add Quick	ADDQB, ADDQW, ADDQD	ADDQi
Add with Carry	ADDCB, ADDCW, ADDCD	ADDCi
Subtract	SUBB, SUBW, SUBD	SUBi
Subtract with Carry [Borrow]	SUBCB, SUBCW, SUBCD	SUBCi
Negate	NEGB, NEGW, NEGD	NEGi
Absolute Value	ABSB, ABSW, ABSD	ABSi
Multiply	MULB, MULW, MULD	MULi
Multiply Extended Integer	MEIB, MEIW, MEID	MEIi
Divide	DIVB, DIVW, DIVD	DIVi
Modulus	MODB, MODW, MODD	MODi
Quotient	QUOB, QUOW, QUOD	QUOi
Remainder	REMB, REMW, REMD	REMi
Divide Extended Integer	DEIB, DEIW, DEID	DEIi
Movement and Conversion		
Move	MOVB, MOVW, MOVD	MOVi
Move Quick	MOVQB, MOVQW, MOVQD	MOVQi
Move with Sign-Extension	MOVXBD,MOVXWD,MOVXBW	MOVXii
Move with Zero-Extension	MOVZBD,MOVZWD,MOVZBW	MOVZii
Comparison		
Compare	CMPB, CMPW, CMPD	CMPi
Compare Quick	CMPQB, CMPQW, CMPQD	CMPQi

Integer operands are binary numbers. An integer operand may be a byte (8 bits), word (16 bits), or double-word (32 bits) in length. Its contents are interpreted as either signed or unsigned.

Unsigned integers range from 0 to 255 (byte), 0 to 65535 (word), and 0 to 4,294,967,295 (double-word). Each bit in an unsigned integer is a value bit, i.e., contributes to the integer's magnitude.

Signed integers are represented in two's-complement form. They range in value from -128 to 127 (byte), -32768 to 32767 (word), and -2,147,483,648 to 2,147,483,647 (double-word) and have the following form:



The most significant bit in a signed integer indicates the sign of the number. A sign bit of zero specifies a positive value in which the remaining bits of the operand are in true binary form. A sign bit of one specifies a negative value, in which the remaining bits hold the two's complement of the absolute value of the operand. The sign bit does not contribute to the integer's magnitude.

The following illustrates a byte, word, and double-word integer and gives the signed and unsigned decimal interpretations for each.

Binary	Signed (Decimal)	Unsigned (Decimal)
10011100	-100	156
1111111011101010	-278	65250
00000000000000000001001000110100	4660	4660

Addition and subtraction operations yield the correct result regardless of whether the operands are interpreted as signed or unsigned. In the Quick instructions, however, one should note that the Quick immediate operand is sign-extended internally before use, and should therefore only be considered signed.

The other integer instructions treat integers as either signed or unsigned, as stated in their individual descriptions in Chapter 5.

Integer Arithmetic

Integer arithmetic is performed to the length specified by the operation length appended to the instruction mnemonic by the programmer. This length may be byte, word or double-word (Section 4.1). Except where noted, the operands of these instructions are both general, meaning that general addressing mode expressions may be used independently to specify the location of each operand.

Addition instructions consist of ADDi, which adds two general operands, and ADDQi (Add Quick), which adds a small value (range -8 to +7) to a single general operand. Extended addition to any length can be performed using the ADDCi instruction, which adds also the contents of the PSR C bit (indicating a carry from a previous addition).

Subtraction (the SUBi instruction) may be modelled as adding together the second operand (the minuend), the one's complement of the first operand (the subtrahend), and the value 1. This definition, using the one's complement, is required to correctly define the overflow and borrow conditions (see "Exceptional Conditions" below). The result is placed in the location of the second operand. Extended subtraction to any length can be performed using the SUBCi instruction, which also subtracts the contents of the PSR C bit (indicating a borrow from a previous subtraction).

Negation (NEGi) and Absolute Value (ABSi) functions are provided. These instructions read a general (source) operand, convert it, and store the result in a second general operand location. Negation is performed by subtracting the source value from zero.

Multiplication is performed according to the standard rules of algebra. The length of the result may be selected as either the same length as the original operands (using the MULi instruction) or double that length (using the MEIi instruction). The MEIi instruction interprets its operands as unsigned integers, making it usable for multiplication to arbitrary length. The distinction between signed and unsigned operands is not relevant to the MULi instruction.

Division is performed according to three separate algorithms. The DIVi instruction divides the second operand by the first, producing as its result the nearest integer which is less than, or equal to, the exact quotient. The QUOi instruction produces the nearest integer whose absolute value is less than, or equal to, the exact quotient. These both interpret their operands as signed values. Note that they differ when the quotient is negative. The DEIi instruction divides a double-length integer (64, 32 or 16 bits) by a single-length divisor, and produces both a quotient and a remainder. It interprets its operands as unsigned for performing extended division; the distinction between the DIVi and QUOi algorithms is therefore irrelevant to this instruction. Remainder instructions are provided for both the DIVi and QUOi algorithms. The MODi (Modulus) instruction performs division according to the DIVi algorithm and produces the remainder as its result. The REMi (Remainder) instruction performs division as per the QUOi instruction and produces the corresponding remainder.

Movement and Conversion

The MOVi instruction moves the first general operand to the second. A variation of this is the MOVQi instruction, which moves a small immediate value (range -8 to +7) into a general operand location.

An integer value can be converted to any greater length while being moved. The conversion for signed integers is provided by the MOVXii instructions, which perform sign-extension, and the conversion for unsigned integers is provided by the MOVZii instructions, which perform zero-extension.

Comparison

Integer comparison instructions compare two operands and set the PSR Z, N and L bits to form a condition code. This condition code can be tested by subsequent instructions for program control or saved to generate operands for Boolean computations.

The CMPi instruction compares two general operands. The CMPQi instruction compares a general operand to a small immediate value (range -8 to +7).

The contents of the PSR Z and N bits indicate the result of comparing the operands as signed integers. The Z bit indicates equality when set. The N bit, when set, indicates that the first operand is greater than the second.

The contents of the PSR Z and L bits indicate the result of comparing the operands as unsigned integers. The Z bit indicates equality when set. The L bit, when set, indicates that the first operand is greater than the second.

Exceptional Conditions

Three exceptional conditions may occur in integer operations. These are a carry (or borrow), an overflow, or attempted division by zero.

Carry and borrow events are signaled in the Processor Status register C bit (Section 2.2). When an addition instruction is executed, the occurrence of a carry out of the most significant bit position (bit 7, 15, or 31, depending on the selected operation length, Section 4.1) constitutes a "Carry" condition, and is indicated by setting the PSR C bit. If no carry occurs, the PSR C bit is cleared. When a subtraction instruction is executed, the lack of a carry out of the most significant bit position constitutes a "Borrow" condition, and the PSR C bit is set to indicate this exceptional condition. If a carry does occur, the PSR C bit is cleared. The result delivered follows the standard rules of binary two's-complement arithmetic, regardless of the occurrence of a carry or borrow condition.

Overflow events from addition and subtraction are signaled in the Processor Status Register F bit (Section 2.2). If the carry into the sign bit position and the carry out of the sign bit position do not agree, this constitutes an "overflow" condition, indicating that the correct result would be too great in magnitude to represent as a signed integer in the number of bits selected as the operation length (Section 4.1). If an overflow occurs in executing an addition or subtraction instruction, the PSR F bit is set, otherwise it is cleared. The result delivered follows the standard rules of binary two's-complement arithmetic (including alteration of the sign bit), regardless of the occurrence of an overflow.

Attempted division by zero always causes a trap, Trap(DVZ). This trap can occur in the DIVi, MODi, QUOi, REMi and DEIi instructions. A trapped instruction delivers no result, neither to the destination operand location nor to the PSR.

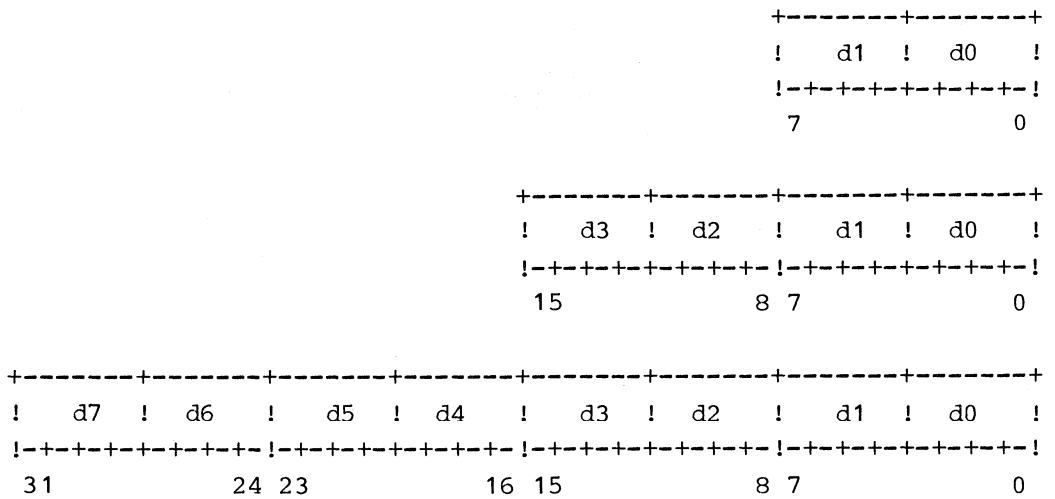
See Chapter 6 for details of trap handling.

3.2 Packed Decimal Instructions

Packed Decimal instructions add and subtract packed decimal operands. There are two Packed Decimal instructions:

Instruction	Mnemonic Forms	Index
Add Packed Decimal	ADDPB, ADDPW, ADDPD	ADDPi
Subtract Packed Decimal	SUBPB, SUBPW, SUBPD	SUBPi

A packed decimal operand consists of two, four, or eight binary-coded decimal (BCD) digits stored in a byte, word, or double-word, respectively. A BCD digit is a 4-bit field whose value is within the range 0 to 9, encoded as binary 0000 to 1001, respectively. Each byte contains two BCD digits as illustrated below. Digit d0 is the least-significant digit.



Packed Decimal instructions operate on two general operands. Both operands are interpreted as unsigned numbers. The ADDPi instruction places the sum of the two operands, plus the contents of the PSR C bit, into the second operand location. The SUBPi instruction subtracts the first operand from the second, subtracting also the contents of the PSR C bit, and places the result into the second operand location. Incorporation of the PSR C bit into the result facilitates use of these instructions in performing packed decimal calculations to arbitrary length.

Decimal subtraction can be modeled as adding the ten's complement of the subtrahend to the minuend.

Both operands must contain only legal BCD digits. If either operand contains digits which are not legal, the result value is undefined, and the setting of the PSR C bit is undefined.

Exceptional Conditions

A decimal carry or borrow condition can occur from Packed Decimal instructions. Decimal carry and borrow events are signaled in the Processor Status register C bit (Section 2.2).

When the ADDPi instruction is executed, the occurrence of a carry out of the most-significant digit position constitutes a "carry" condition, and is indicated by the CPU by setting the PSR C bit. This indicates that the sum is too large to be held as a Packed Decimal number in the length of the original operands. The result produced is the least-significant portion of the entire result.

If no carry occurs, the PSR C bit is cleared.

When the SUBPi instruction is executed, the lack of a carry out of the most significant digit position constitutes a "borrow" condition, and the PSR C bit is set to indicate this. A borrow condition indicates that a high-order "1" digit has been assumed to the left of the most-significant minuend digit in order to produce a positive result.

If a carry does occur from subtraction, the PSR C bit is cleared.

3.3 Floating-Point Instructions

Floating-Point instructions operate on floating-point numbers. Included also in this group are the instructions which load and store the Floating-Point Status register (FSR). The following is a list of the Floating-Point instructions:

Instruction	Mnemonic Forms	Index
Add Floating	ADDF, ADDL	ADDf
Subtract Floating	SUBF, SUBL	SUBf
Multiply Floating	MULF, MULL	MULf
Divide Floating	DIVF, DIVL	DIVf
Negate Floating	NEGF, NEGL	NEGf
Absolute Value Floating	ABSF, ABSL	ABSf
Compare Floating	CMPF, CMPL	CMPf
Move Floating	MOVF, MOVL	MOVf
Move Long Floating to Floating	MOVLF	MOVLF
Move Floating to Long Floating	MOVFL	MOVFL
Move Integer to Floating	MOVBF, MOVWF, MOVDF, MOVBL, MOVWL, MOVDL	MOVif
Round Floating to Integer	ROUNDFB, ROUNDFW, ROUNDFD, ROUNDLB, ROUNDLW, ROUNDLD	ROUNDfi
Truncate Floating to Integer	TRUNCFB, TRUNCFW, TRUNCFD, TRUNCLB, TRUNCLW, TRUNCLD	TRUNCfi
Floor Floating to Integer	FLOORFB, FLOORFW, FLOORFD, FLOORLB, FLOORLW, FLOORLD	FLOORfi
Load FSR	LFSR	LFSR
Store FSR	SFSR	SFSR

Floating-point arithmetic operations are performed by the ADDf, SUBf, MULf and DIVf instructions. The NEGf and ABSf instructions move the negative or the absolute value of their first operand to the second operand location. The CMPf instruction compares two floating-point values, setting the PSR condition codes as per the CMPi (integer compare) instruction. The MOVf instruction moves a floating-point value.

The full range of conversions are provided; between floating-point types, and between any integer and floating-point types. Conversion from floating-point to integers can be performed by rounding to nearest (ROUNDfi), toward zero (TRUNCfi) or toward negative infinity (FLOORfi).

The LFSR and SFSR instructions load and store the FSR, which holds mode and status information pertaining to floating-point operations (Section 2.4.2).

3.3.1 Floating-Point Operand Formats

The Series 32000 Floating-Point instruction set operates on two floating-point data types: single precision (32 bits) and double precision (64 bits). Floating-point instruction mnemonics use the operation length suffix F (Floating) to specify the single precision data type and the suffix L (Long Floating) to specify the double precision data type.

A floating-point number is divided into three fields as shown in Figure 3-1.

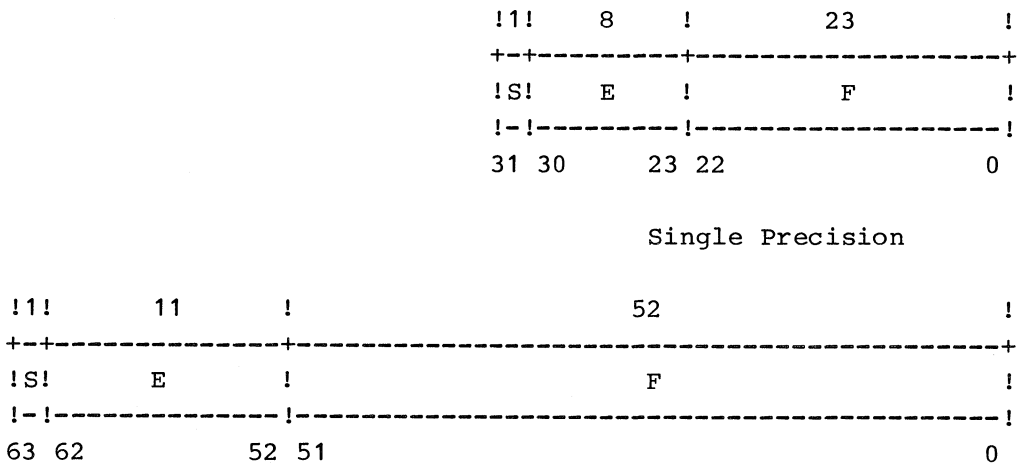


Figure 3-1 Floating-Point Operand Formats

The F field is the fractional portion of the represented number. The binary point is assumed to be immediately to the left of the most-significant bit of the F field, with an implied 1 bit to the left of the binary point. Thus, the F field represents values from 1.0 (inclusive) to 2.0 (exclusive) as shown in Table 3-1.

Table 3-1 SAMPLE F FIELDS

F Field	Binary Value	Decimal Value
000...0	1.000...0	1.000...0
010...0	1.010...0	1.250...0
100...0	1.100...0	1.500...0
110...0	1.110...0	1.750...0
	↑ Implied	

The E field holds an unsigned number which gives the binary exponent of the represented number. The value in the E field is biased; that is, a constant bias value must be subtracted from the value in the E field in order to obtain the true exponent. This bias value is 011...11 (binary), which is either the value 127 (in single precision) or 1023 (in double precision). Thus, the true binary exponent can be either positive or negative, as shown in Table 3-2.

Table 3-2 Sample E Fields

E Field	F Field	Represented Value
011...110	100...0	$1.5 * 2^{-1} = 0.75$
011...111	100...0	$1.5 * 2^{0} = 1.50$
100...000	100...0	$1.5 * 2^{1} = 3.00$

NOTE: Two forms of the E field represent special values, and are not interpreted as binary exponent values. 11...11 represents a value which is a Reserved operand (Section 3.3.4). 00...00 represents the value Zero (Section 3.3.3) if the F field is also all zeroes, otherwise the represented value is a Reserved operand.

The S bit indicates the sign of the operand: 0 for positive and 1 for negative. Floating-point numbers are represented in sign-magnitude form, such that only the S bit is complemented in order to change the sign of the represented number.

3.3.2 Normalized Numbers

Normalized numbers are numbers in floating-point format, where the E field is neither all zeroes nor all ones.

The value represented by a normalized number is determined by the formula:

$$(-1)^{S} * 2^{(E-Bias)} * 1.F$$

The ranges of normalized numbers are given in Table 3-3.

Table 3-3 NORMALIZED FLOATING-POINT RANGES

	Single Precision	Double Precision
Most Positive	$2^{127} * (2 - 2^{-23})$ $= 3.40282346 * 10^{38}$	$2^{1023} * (2 - 2^{-52})$ $= 1.7976931348623157 * 10^{308}$
Least Positive	2^{-126} $= 1.17549436 * 10^{-38}$	2^{-1022} $= 2.2250738585072014 * 10^{-308}$
Least Negative	$-(2^{-126})$ $= -1.17549436 * 10^{-38}$	$-(2^{-1022})$ $= -2.2250738585072014 * 10^{-308}$
Most Negative	$-2^{127} * (2 - 2^{-23})$ $= -3.40282346 * 10^{38}$	$-2^{1023} * (2 - 2^{-52})$ $= -1.7976931348623157 * 10^{308}$

NOTE: The values given are extended one full digit beyond their represented accuracy to help in generating rounding and conversion algorithms.

3.3.3 Zero

There are two representations for zero -- a positive form and a negative form. Positive zero has all-zero F and E fields, and its S bit is zero. Negative zero also has all-zero F and E fields, but its sign bit is one. In spite of these differences, the two zeroes are considered equal to each other when compared using the CMPf instruction.

3.3.4 Reserved Operands

The proposed IEEE Standard for Binary Floating Point Arithmetic (IEEE Task P754) provides for certain exceptional forms of floating-point operands. The Series 32000 hardware currently treats these forms as reserved operands. The reserved operands are:

- Positive and Negative Infinity
- Not-a-Number (NaN) values
- Denormalized numbers

Both Infinity and NaN values have all ones in their E fields. Denormalized numbers have all zeroes in their E fields and non-zero values in their F fields.

The Series 32000 hardware causes an Invalid Operation trap (Section 3.3.7) if it receives a reserved operand, unless the instruction being executed is a simple MOVf instruction (move without conversion). The Series 32000 hardware does not generate reserved operands as results of floating-point calculations. The trapping mechanism used in the Series 32000 family allows handling of these operand forms transparently in software.

3.3.5 Integers

Some floating-point instructions perform conversions between integer and floating-point data types. Integers are accepted and generated as two's complement values of byte, word or double-word length, as specified in the conversion instruction.

3.3.6 Memory Representations

Floating-point operands are stored in memory with the least-significant byte at the lowest address, except in the Immediate addressing mode. In this mode, the operand is held within the instruction format with the most-significant byte at the lowest address.

3.3.7 Floating-Point Traps

Trap (UND)

The Floating-Point instruction set is made available to a Series 32000-based system with an NS32081 Floating-Point Unit by setting the F bit in the CFG register (Section 2.3). If the CFG F bit is not set, any floating-point instruction causes the Undefined Instruction trap, Trap (UND). See Chapter 6 for further details. In systems without floating-point hardware, Trap (UND) can be used to transfer control to floating-point emulation software.

Trap (FPU)

Any exceptional conditions encountered during the execution of a floating-point instruction will cause a floating-point trap. This trap is labeled Trap (FPU) and uses the fourth entry (entry #3) of the Interrupt Dispatch Table (Chapter 6).

The following are true for any floating-point instruction causing Trap (FPU):

1. The status fields of the FSR are updated before trapping.

2. No other result is delivered, neither to the destination operand location nor to the Processor Status Register (PSR).

3. The return address pushed onto the Interrupt Stack is the address of the first byte of the trapped instruction. This allows software analysis or emulation of the trapped instruction, or re-execution after the exception has been logged.

For further details of trap service, see Chapter 6.

The conditions which cause Trap (FPU) are:

1. Underflow. A non-zero floating-point result is too small in magnitude to be represented as a normalized floating-point number in the format of the destination operand. This condition is always reported in the FSR TT field and UF bit, but causes a Trap (FPU) only if the FSR UEN bit is set. If the UEN bit is not set, a result of Positive Zero is produced, and no trap occurs.

2. Overflow. A result (either floating-point or integer) of a floating-point instruction is too great in magnitude to be held in the format of the destination operand. Note that rounding, as well as calculations, can cause this condition.

3. Divide by Zero. An attempt has been made to divide a non-zero floating-point number by zero. Dividing zero by zero is considered an Invalid Operation instead (below). Note that the trap caused by this condition is still Trap (FPU) and not Trap (DVZ), which is caused only by integer instructions.

4. Illegal Instruction. Two undefined floating-point instruction forms cause Trap (FPU) rather than Trap (UND). The binary formats causing this trap are:

xxxxxxxxxx0011xx10111110
xxxxxxxxxx1001xx10111110

5. Invalid Operation. One of the floating-point operands of a floating-point instruction is a Reserved operand (Section 3.3.4), or an attempt has been made to divide zero by zero using the DIVf instruction.

6. Inexact Result. The result (either floating-point or integer) of a floating-point instruction cannot be represented exactly in the format of the destination operand, and a rounding step must alter it to fit. This condition is always reported in the FSR TT field and IF bit unless any other exceptional condition has occurred in the same instruction. In this case, the TT field always contains the code for the other exception and the IF bit is not altered. A Trap (FPU) is caused by this condition only if the FSR IEN bit is set; otherwise the result is rounded and delivered, and no trap occurs.

3.4 Logical Instructions

Logical instructions perform masking, shifting and Boolean arithmetic operations. The following table lists the logical instructions:

Instruction	Mnemonic Forms	Index
Arithmetic		
Logical AND	ANDB, ANDW, ANDD	ANDi
Logical OR	ORB, ORW, ORD	ORi
Bit Clear	BICB, BICW, BICD	BICi
Exclusive OR	XORB, XORW, XORD	XORi
Complement	COMB, COMW, COMD	COMi
Shift		
Arithmetic Shift	ASHB, ASHW, ASHD	ASHi
Logical Shift	LSHB, LSHW, LSHD	LSHi
Rotate	ROTB, ROTW, ROTD	ROTi
Boolean		
Complement Boolean	NOTB, NOTW, NOTD	NOTi
Save Condition as Boolean	ScondB, ScondW, ScondD	Scondi

The arithmetic instructions perform bitwise Boolean arithmetic on byte, word or double-word general operands. The shift instructions perform shifting on byte, word or double-word general operands. The Boolean instructions generate and complement Boolean values.

The ANDi, ORi and XORi instructions perform the bitwise Boolean AND, OR and Exclusive OR functions between two general operands. The BICi instruction performs an AND NOT operation, clearing all bits in the second operand which are set in the first. The COMi instruction moves the bitwise complement of the first operand to the second.

The shift instructions shift their second general operand in the direction and by the magnitude given by the first operand (a positive shift is left, a negative shift is right). The logical shift fills the emptied bit positions with zeroes always. The arithmetic shift fills these locations with zeroes if the shift is to the left, and with the original contents of the sign bit (the most-significant bit) if the shift is to the right. The rotation shift consecutively replaces each bit emptied with the contents of the bit shifted out of the operand.

NOTE: The result generated by shifting an operand by a count which is greater than, or equal to, its length in bits is undefined.

The Boolean instructions generate and handle unpacked Boolean values, defined as integers whose values are interpreted as 0 = False and 1 = True. This definition follows conventions established by several high-level languages which require that True be greater than False when compared and that conversions between Boolean and integer variables generate the above correlation between values.

All of the logical arithmetic instructions perform correct Boolean arithmetic on Boolean values except the COMi instruction. To allow complementing Boolean values (from True to False and vice versa), the NOTi instruction is provided, which complements only the least-significant bit of its first operand, placing the result in the second.

Because Boolean arithmetic often deals with values derived from relational operations (e.g. whether one value is greater than another), the Save Condition (Scondi) instruction is provided, which generates a Boolean value based on a condition code test.

3.5 Bit Instructions

Bit instructions perform or support manipulation of individual bits in General Purpose Registers or memory. The following is a list of the Bit instructions:

Instruction	Mnemonic Forms	Index
Test Bit	TBITB, TBITW, TBITD	TBITi
Set Bit	SBITB, SBITW, SBITD, SBITIB, SBITIW, SBITID	SBITi, SBITIi
Clear Bit	CBITB, CBITW, CBITD, CBITIB, CBITIW, CBITID	CBITi, CBITIi
Invert Bit	IBITB, IBITW, IBITD	IBITi
Find First Set Bit	FFSB, FFSW, FFSD	FFSi
Convert to Bit Pointer	CVTP	CVTP

The TBIT instruction tests a bit by copying its contents to the PSR F bit. The SBIT, CBIT and IBIT instructions test the specified bit, and then either set, clear or invert it. The SBITI and CBITI instructions, in addition, allow testing and either setting or clearing of a bit in an indivisible operation for handling multiprocessor semaphores.

The FFSi and CVTP instructions do not operate on bits, but provide related functions to aid in bit handling. The FFSi instruction scans a byte, word or double-word for a set bit, producing its position as a one-byte offset value. The CVTP instruction generates the bit address of a specified bit.

Bit positions are specified using two general operand specifications: a base and an offset, as in the instruction.

```
TBITi     offset,base
```

The base operand specification is used only to determine a base location (either a memory address or a register) relative to which the bit is to be located, and does not itself reference an operand at that location. The offset is a general operand of byte, word or double-word length, as specified by the operation length selected by the programmer (Section 4.1). It contains a signed integer which specifies the position of the desired bit relative to bit 0 of the location specified as the base.

If the base is specified as a General Purpose register, the offset must be within the range 0 to 31, inclusive. If the offset is outside this range, the location of the bit is undefined.

If the base is specified as a memory address, the offset specifies a bit in memory.

Both positive and negative offsets are allowed and meaningful. An offset of 0 specifies bit 0 of the byte at the base address. An offset of 8 specifies bit 0 of the byte at the next higher address. An offset of -1 specifies bit 7 of the byte at the next lower address, and an offset of -8 specifies bit 0 of the byte at the next lower address.

The maximum range of a double-word offset is -2,147,483,648 to +2,147,483,647 bits, corresponding to an addressing range of -268,435,456 to +268,435,455 bytes from the specified base. Note that this is considerably greater than the memory space currently implemented.

If the offset operand specifies a bit outside the memory space, the location of the bit is undefined. See Section 2.6.1 for considerations of memory size.

The address of the byte containing the desired bit is formally defined as

EA(base) + (offset DIV 8)

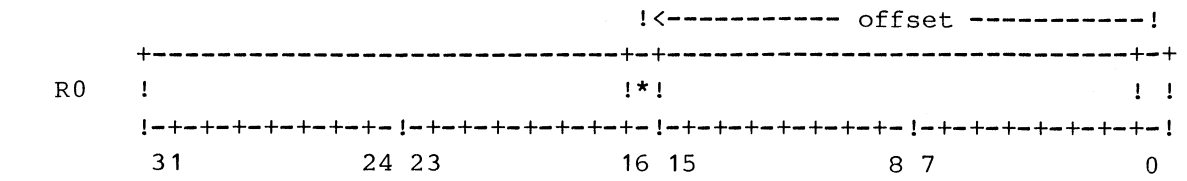
where "EA(base)" is the effective address calculated from the base operand specification and "offset DIV 8" is the nearest integer less than or equal to offset/8 (as per the DIVi instruction). The bit number of the desired bit is computed as

offset MOD 8

where MOD is the modulus function (as per the MODi instruction).

The following examples illustrate the interpretations of various bit specifications:

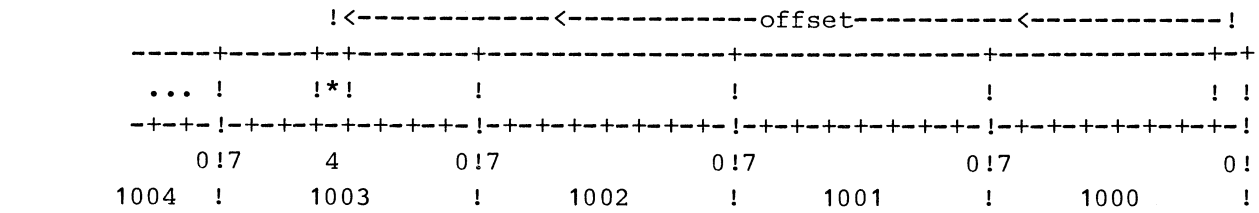
Example 1:



Offset : 16 Base : R0

Interpreted as bit 16 of register R0.

Example 2:

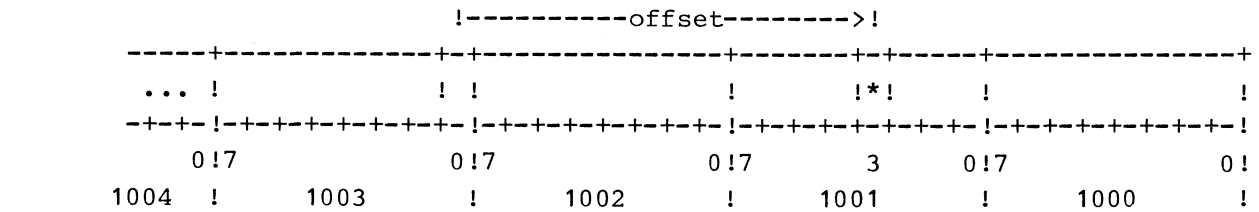


Offset: +28 EA(Base): 1000

Interpreted as bit 4 of the byte at address 1003.

In this example, the address of the byte containing the desired bit is 1000 + (28 DIV 8), or 1003, since 28 DIV 8 = 3. The bit number within this byte is 28 MOD 8, or 4.

Example 3:



Offset: -13 EA(Base): 1003

Interpreted as bit 3 of the byte at address 1001.

In this example, the address of the byte containing the desired bit is 1003 + (-13 DIV 8), or 1001, since -13 DIV 8 = -2. The bit number within this byte is -13 MOD 8, or 3. If these results look confusing, consult again the definitions of the DIV and MOD operations given above.

3.6 Bit Field Instructions

Bit Field instructions copy information to and from unaligned fields in General Purpose Registers or memory. The following is a list of the Bit Field instructions:

Instruction	Mnemonic Forms	Index
Extract Field	EXTB, EXTW, EXTD	EXTi
Extract Field Short	EXTSB, EXTSW, EXTSD	EXTSi
Insert Field	INSB, INSW, INSD	INSi
Insert Field Short	INSSB, INSSW, INSSD	INSSi

Extract instructions read a bit field and place it into a byte, word, or double-word general operand, right-justified. Insert instructions replace a bit field from aligned information in a general operand. A bit field may be one to 32 bits in length.

A bit field is fully specified by the position of its least-significant bit and its length in bits. The position of the least-significant bit is specified as in the Bit instructions (Section 3.5), using a general operand specification for the base and an offset contained either in a General Purpose Register or (in the "Short" forms of these instructions) in an immediate constant. The length of the field is specified as an immediate constant, which must specify a length in the range of 1 to 32 bits, inclusive. The interpretation of any length specified outside this range is undefined.

The general bit field instructions (EXTi and INSi) allow a 32-bit offset value to be dynamically specified in a General Purpose Register, supporting the indexing necessary to access structures such as Pascal packed arrays. The "Short" bit field instructions (EXTSi and INSSi) eliminate the overhead of loading a register when the offset is fixed, as is commonly the case in accessing structures such as Pascal packed records.

If the base is specified as a General Purpose Register, the bit field is in that register. The offset must be within the range 0 to 31, and the entire bit field must be contained within the specified register, otherwise the location of the bit field is undefined.

If the base is specified as a memory address, the offset specifies a bit in memory as the least-significant bit of the field. Both positive and negative offsets are allowed and meaningful, as in Bit instructions (Section 3.5). If the offset specifies a bit outside the memory space, or if it causes the bit field to extend outside the memory space, the location of the bit field is undefined. See Section 2.6.1 for considerations of memory size.

As in the Bit instructions, the address of the byte containing the least-significant bit of the field is defined as

EA(Base) + (offset DIV 8)

where "EA(Base)" is the effective address calculated from the base operand specification and "offset DIV 8" is the nearest integer less than, or equal to, offset/8 (as per the DIVi instruction). The bit number of the least-significant bit in the field is computed as

offset MOD 8

where MOD is the modulus function (as per the MODi instruction).

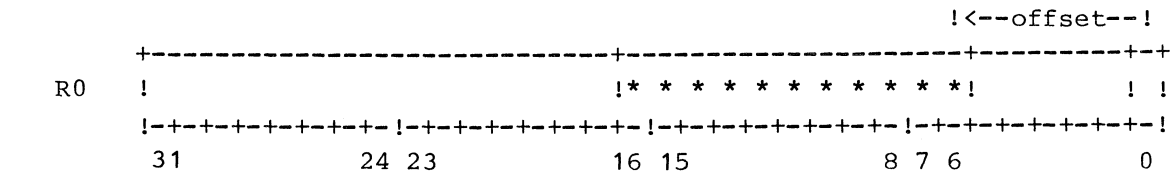
NOTES:

1. The current implementation of bit field instructions places an alignment restriction on bit fields greater than 25 bits in length. This restriction is imposed due to the fact that a field in memory is accessed in a double-word transfer starting at the byte containing the least-significant bit of the field. A bit field in memory must be composed of bits from no more than four contiguous bytes. For a field of 25 bits or less, this imposes no restriction on alignment, as it is impossible for such a field to span more than four bytes.

2. Regardless of the length of a bit field in memory, it is always accessed by Bit Field instructions as a double-word starting with the byte which contains the least-significant bit of the field. The Extract instructions read a full double-word, and the Insert instructions read, modify and rewrite a full double-word. These instructions can therefore cause a page fault in memory-managed systems if the field is close to the end of a page. In multiprocessor systems, care should be taken to ensure that the processors do not attempt to modify adjacent fields simultaneously.

The following examples illustrate how a bit field is located in a register and in memory:

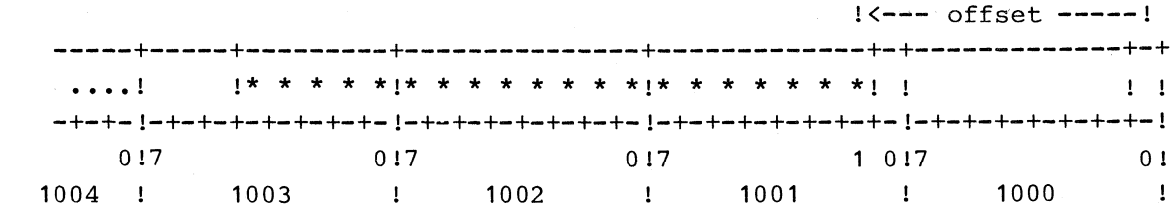
Example 1:



Base: R0 Offset: 6 Length: 11

Interpreted as an 11-bit field in register R0 starting with bit 6.

Example 2:

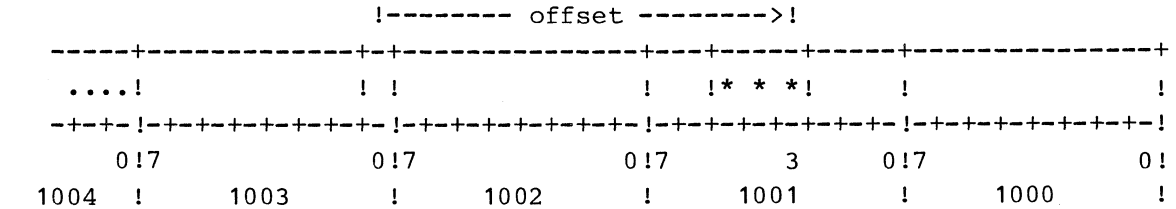


EA(Base): 1000 Offset: +9 Length: 20

Interpreted as a 20-bit field starting at bit 1 of address 1001.

In this example, the address of the byte containing the least-significant bit of the field is 1000 + (9 DIV 8), or 1001. The bit number of the first field bit within that byte is 9 MOD 8, or 1.

Example 3:



EA(Base): 1003 Offset: -13 Length: 3

Interpreted as a 3-bit field starting at bit 3 of address 1001.

In this example, the address of the byte containing the least-significant bit of the field is 1003 + (-13 DIV 8), or 1001, since -13 DIV 8 = -2. The bit number of the first field bit in that byte is -13 MOD 8, or 3. If these results look confusing, consult again the definitions of the DIV and MOD operations given above.

3.7 String Instructions

String instructions operate on strings of integer elements. The following is a list of the String instructions:

Instruction	Mnemonic Forms	Index
Move String	MOVSB, MOVSW, MOVSD	MOVSi
Move String, Translating	MOVST	MOVST
Compare Strings	CMPSB, CMPSW, CMPSD	CMPSi
Compare Strings, Translating	CMPST	CMPST
Skip String	SKPSB, SKPSW, SKPSD	SKPSi
Skip String, Translating	SKPST	SKPST

A string is a sequence of integer elements, all of the same length, stored in consecutive memory locations. Elements of a string may be bytes, words, or double-words as specified by the operation length (Section 4.1), except when the Translating form (above) is used, in which case the elements must be bytes.

String instructions operate on either one or two strings. These strings are designated String 1 and String 2. The MOVS instructions copy elements from String 1 to String 2. The CMPS instructions compare String 1 elements to the corresponding String 2 elements. The SKPS instructions scan elements of String 1, without using a String 2.

String locations and length are specified by the General Purpose registers R0, R1, and R2. Before instruction execution, the registers must be set to the following:

R0 -- the maximum number of elements to be processed
R1 -- the address of the first element of String 1
R2 -- the address of the first element of String 2 (except for SKPS, which does not use or modify R2)

NOTE: The number of elements processed is undefined if register R0 contains a negative number.

String instructions process the elements of the string(s) one at a time until a specified termination condition is reached. After each element is processed, the instructions modify the 32-bit contents of registers R0, R1, and R2 so that they contain the following values:

R0 -- the number of elements left to be processed (old contents minus one)
R1 -- the address of the next element of String 1
R2 -- the address of the next element of String 2 (except in SKPS)

If the resulting value in R0 is zero, the instruction terminates. The contents of register R2 always remain unchanged by the SKPS instruction.

Options

String instructions have the following options:

- Translation (T)
- Backward (B)
- Until Match (U)
- While Match (W)

Additional information required by these options is specified in General Purpose registers R3 and R4, as follows:

R3 -- the address of a translation table, required if the Translation option is specified

R4 -- a termination value, required if the Until Match or While Match option is specified

Registers R3 and R4 remain unchanged by the instruction.

The Translation option causes a string instruction to translate each String 1 element before using it. String instructions with the Translation option operate on 1-byte elements only, and because of this the Translation option is specified as a mnemonic suffix "T" replacing the operation length suffix.

Translation is performed by using the String 1 element value as an unsigned index into a translation table, whose base address is taken from register R3. A byte is read from this table location, and is used in place of the original String 1 element.

The Backward option causes a string instruction to reverse its direction, processing string elements from successively lower memory addresses instead of successively higher addresses. This means that registers R1 and R2 are decremented by the element length after each element is processed instead of being incremented. The Backward option is specified in assembly language by listing the letter B in the instruction as an operand. When used in conjunction with the Until Match or While Match option, it must be separated with a comma.

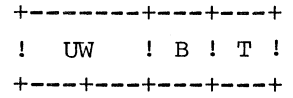
The Until Match and While Match options specify a termination condition based on whether the contents of each String 1 element match the contents of register R4 (after translation, if that option is also specified). In order to distinguish this termination condition from any other, the PSR F bit is set to 1 before termination. The Until Match and While Match options are mutually exclusive.

If the Until Match option is specified, the instruction terminates as soon as the current value matches R4. This option is specified in assembly language by listing the letter U in the instruction as an operand.

If the While Match option is specified, the instruction terminates as soon as the current value does not match R4. This option is specified in assembly language by listing the letter W in the instruction as an operand.

Option Encoding

Each string instruction contains a 4-bit field defining which options are specified. The field has the following form:



The 1-bit T field defines the state of the Translation option. If the field is 1, the Translation option is in effect; otherwise, the option is not in effect. If the T bit is set, the operation length field (i) must contain binary 00 (Byte).

The 1-bit B field defines the state of the Backward option. If the field is 1, the option is in effect; otherwise, the option is not in effect.

The 2-bit UW field defines the state of the Until and While options, as given below:

00	neither option
01	While Match
10	(reserved)
11	Until Match

Interrupts During String Instructions

String instructions are interruptible. If an interrupt is asserted during a String instruction, the CPU first finishes processing the current string element. It then saves the address of the String instruction as the return address and passes control to the interrupt service procedure. When the interrupt service procedure returns, the String instruction is re-executed, but because the registers have been updated this has the effect of continuing string processing from the point where the instruction was interrupted. Note that the interrupt service procedure must follow the standard practice of restoring all registers used before returning.

Termination Conditions

A string instruction terminates for one of the following reasons:

1. The limit count originally specified in register R0 has been decremented to zero, or was zero at the beginning of the instruction.

2. The CMPS instruction has found a pair of string elements which are unequal and has, therefore, determined which string has the greater value.

3. The Until Match or While Match option is in effect and the string instruction has found an element in String 1 which meets the specified termination condition.

When a string instruction terminates due to its limit count, the resulting state of the machine is as follows:

PSR - bit F = 0. If a CMPS instruction terminates for this reason, then also PSR bits Z = 1, N = 0, L = 0.
R0 -- contains 0.
R1 -- contains the address of the next unprocessed String 1 element.
R2 -- contains the address of the next unprocessed String 2 element (except in SKPS).

When a CMPS instruction finds an unequal pair of string elements, the resulting state of the machine is:

PSR - bits F = 0 and Z = 0. The N and L bits indicate the relation between the two unequal string elements.
R0 -- contains the number of element pairs left to be processed (this includes the element pair which caused termination).
R1 -- contains the address of the String 1 element which caused termination.
R2 -- contains the address of the String 2 element which caused termination.

Whenever the Until Match or While Match option terminates execution of a string instruction, the resulting machine state is:

PSR - bit F = 1. If a CMPS instruction terminates for this reason, then also PSR bits Z = 1, N = 0, L = 0.
R0 -- contains the number of elements left to be processed (this includes the element which caused termination).
R1 -- contains the address of the element in String 1 which caused termination
R2 -- contains the address of the element in String 2 which corresponds to the String 1 element which caused termination (except in SKPS)

The contents of registers R3 and R4 always remain unchanged.

Detailed Sequences

Table 3-4 below gives the detailed execution sequences followed by the string instructions. A temporary holding location within the processor is referenced by the name "TEMP".

Table 3-4 Execution Sequences

	CMPS	MOVS	SKPS
1	In the PSR, set bits Z=1, N=0, L=0. If R0 = 0, set the PSR F bit to 0 and terminate the instruction.	If R0 = 0, set the PSR F bit to 0 and terminate the instruction.	If R0 = 0, set the PSR F bit to 0 and terminate the instruction.
2	Read the current String 1 element (address in R1) from memory into TEMP.	Read the current String 1 element (address in R1) from memory into TEMP.	Read the current String 1 element (address in R1) from memory into TEMP.
3	If the Translation option is selected, then zero-extend TEMP from 8 bits to 32 bits and add it to the contents of R3, generating the address of a translation table entry. Read a byte from this memory location and place it into TEMP.	If the Translation option is selected, then zero-extend TEMP from 8 bits to 32 bits and add it to the contents of R3, generating the address of a translation table entry. Read a byte from this memory location and place it into TEMP.	If the Translation option is selected, then zero-extend TEMP from 8 bits to 32 bits and add it to the contents of R3, generating the address of a translation table entry. Read a byte from this memory location and place it into TEMP.
4	If the Until Match or While Match option is specified, then compare TEMP to R4, interpreting both as integers of the size specified by the operation length. If the Until Match option is specified, and TEMP and R4 are equal, then set the PSR F bit to 1 and terminate the instruction. If the While Match option is specified, and TEMP and R4 are unequal, then set the PSR F bit to 1 and terminate the instruction.	If the Until Match or While Match option is specified, then compare TEMP to R4, interpreting both as integers of the size specified by the operation length. If the Until Match option is specified, and TEMP and R4 are equal, then set the PSR F bit to 1 and terminate the instruction. If the While Match option is specified, and TEMP and R4 are unequal, then set the PSR F bit to 1 and terminate the instruction.	If the Until Match or While Match option is specified, then compare TEMP to R4, interpreting both as integers of the size specified by the operation length. If the Until Match option is specified, and TEMP and R4 are equal, then set the PSR F bit to 1 and terminate the instruction. If the While Match option is specified, and TEMP and R4 are unequal, then set the PSR F bit to 1 and terminate the instruction.
5	Compare TEMP to the contents of the current String 2 location (address in R2) and update PSR bits Z, N and L to reflect the result. If the resulting Z bit is zero (meaning not equal), then set the PSR F bit to 0 and terminate the instruction.	Write TEMP to the current String 2 location (address in R2).	Do nothing; continue to Step 6.
6	If the Backward option is specified, decrement R1 and R2 by the length in bytes specified by the operation length. Otherwise, increment R1 and R2 by this amount.	If the Backward option is specified, decrement R1 and R2 by the length in bytes specified by the operation length. Otherwise, increment R1 and R2 by this amount.	If the Backward option is specified, decrement R1 by the length in bytes specified by the operation length. Otherwise, increment R1 by this amount.
7	Decrement R0 by 1.	Decrement R0 by 1.	Decrement R0 by 1.
8	If an interrupt is pending, service it here. Otherwise, go to Step 1.	If an interrupt is pending, service it here. Otherwise, go to Step 1.	If an interrupt is pending, service it here. Otherwise, go to Step 1.

3.8 Block Instructions

Block instructions move and compare byte, word, and double-word elements stored in contiguous blocks of memory. There are two block instructions:

Instruction	Mnemonic Forms	Index
Move Multiple	MOVMB, MOVMW, MOVMD	MOVMi
Compare Multiple	CMPMB, CMPMW, CMPMD	CMPMi

A block is a small string (16 bytes or less) of integers.

Block instructions differ from their string counterparts in three major ways:

1. They require no overhead in setting up registers, as both block operands are general.

2. They are not interruptible.

3. They are limited to blocks of 16 bytes or less so that they do not adversely affect interrupt latency.

Block instructions have three operands: block1, block2, and length. The MOVMi instruction copies block1 to block2. The CMPMi instruction compares the elements of block1 to the corresponding block2 elements, indicating in PSR bits Z, N and L which block contains the greater value, or whether they are equal.

Block1 and block2 are general operands which must be in memory (access class addr, Section 4.2.1).

The length operand is an immediate value which specifies the length of each block. In assembly language, length is specified as the number of elements (bytes, words or double-words) in the block. (This is not the value which is encoded in the binary form of the instruction.) Since a block must contain at least one byte and no more than 16 bytes, the range of values for length depends on the instruction's operation length suffix (B, W, or D: Section 4.1) as shown by the following:

Operation Length Suffix	length
B	1 to 16
W	1 to 8
D	1 to 4

In the binary form of the instruction, the block length is encoded in a displacement field and appended to the basic instruction. The displacement field contents are to be computed from the specified length value as

(length - 1) * i

where i is the element size in bytes: 1 (for B), 2 (for W), or 4 (for D).

NOTES:

1. The two block operands of the MOVMi instruction must not overlap. If they do overlap, the resulting values in the destination block are undefined.

2. If the binary contents of the length operand differ from those values which can be derived from the expression above, the length of the blocks is undefined.

3.9 Array Instructions

Array instructions operate in conjunction with the Scaled Indexing addressing mode option (Section 4.4.9) to support random accesses into single- and multi-dimensional arrays. The following is a list of the array instructions:

Instruction	Mnemonic Forms	Index
Bounds Check	CHECKB, CHECKW, CHECKD	CHECKi
Calculate Index	INDEXB, INDEXW, INDEXD	INDEXi

An array consists of a number of elements of the same length, stored in a contiguous block of memory. An array can be of a single dimension (i.e., a vector) or of multiple dimensions (i.e., a matrix). Individual elements in an array are accessed using one subscript or index expression per dimension.

The CHECKi instruction performs a bounds check on any general operand, checking whether its value is within the range specified by a pair of values in another general operand. If so, it zero-adjusts the value by subtracting the lower bound from it, and places the result in any specified General Purpose Register. If not, it indicates an error in the PSR F bit, which can be used either as a branch condition or to cause a trap (see the FLAG instruction). If the value being checked is an index into a single-dimensional array, the result placed in the register is directly usable with Scaled Indexing to access the indicated array element.

The INDEXi instruction is used for accesses into multidimensional arrays. Its purpose is to calculate a single 1-dimensional index based on the values of the indexes (one per dimension) by which the desired element is specified. The order in which the indexes are incorporated into the result depends on the scheme used for ordering the array elements in memory.

Depending on the high-level language, array storage ordering generally follows one of two schemes. Row major ordering, the most popular, and typical of the Pascal and C languages, is shown in Table 3-5. Column major ordering, typical of FORTRAN, is shown in Table 3-6. Note that in row major ordering it is the rightmost index which is incremented with consecutive element addresses, and in column major ordering it is the leftmost.

Table 3-5 Row Major Ordering

Pascal array declaration:

```
VAR A: ARRAY[1..2,1..3,1..2]
     OF INTEGER;
```

Element size: 4 bytes
Base address: 1000 (Hex)

Array Element	Address (Hex)
A [1,1,1]	1000
A [1,1,2]	1004
A [1,2,1]	1008
A [1,2,2]	100C
A [1,3,1]	1010
A [1,3,2]	1014
A [2,1,1]	1018
A [2,1,2]	101C
A [2,2,1]	1020
A [2,2,2]	1024
A [2,3,1]	1028
A [2,3,2]	102C

Table 3-6 Column Major Ordering

FORTRAN array declaration:

```
INTEGER A(2,3,2)
```

Element size: 4 bytes
Base address: 1000 (Hex)

Array Element	Address (Hex)
A (1,1,1)	1000
A (2,1,1)	1004
A (1,2,1)	1008
A (2,2,1)	100C
A (1,3,1)	1010
A (2,3,1)	1014
A (1,1,2)	1018
A (2,1,2)	101C
A (1,2,2)	1020
A (2,2,2)	1024
A (1,3,2)	1028
A (2,3,2)	102C

Note that the same memory location is referenced by the Pascal index sequence [I,J,K] and the FORTRAN index sequence (K,J,I).

The general expression for the one-dimensional index generated to access either A[I,J,K, ... ,Z] in Pascal or A(Z, ... ,K,J,I) in FORTRAN is:

```
(...((Ia*Dj+Ja)*Dk+Ka)*...)*Dz+Za
```

where Dj, Dk, ... , Dz are the lengths of A along the J, K, ... , and Z dimensions, respectively, and the values Ia, Ja, Ka, ... , Za
are the index values, zero-adjusted by the CHECKi instruction (by subtracting their lower bounds).

The INDEXi instruction implements one step of the evaluation of this expression from the inside out, by providing the function

```
accum = accum * (length+1) + index
```

where accum is any register (R0-R7), used in consecutive INDEXi instructions as an accumulator location,
index is the current index value being processed, and
length is a general operand containing the current dimension length minus 1 (so that it always matches the size of the index operand).

3.10 Processor Control Instructions

Processor control instructions control the sequence of program execution. These instructions provide conditional and unconditional branches, calls to and returns from local and external procedures, and generation and returns from traps and interrupts. The following is a list of the processor control instructions:

Instructions	Mnemonic Forms	Index
Branches		
Jump	JUMP	JUMP
Conditional Branch	Bcond	Bcond
Unconditional Branch	BR	BR
Case Branch (Multiway)	CASEB, CASEW, CASED	CASEi
Add, Compare and Branch	ACBB, ACBW, ACBD	ACBi
Local Procedure Calls/Returns		
Jump to Subroutine	JSR	JSR
Branch to Subroutine	BSR	BSR
Return from Subroutine	RET	RET
External Procedure Calls/Returns		
Call External Procedure	CXP	CXP
Call External Procedure with Descriptor	CXPD	CXPD
Return from External Procedure	RXP	RXP
Explicit Trap Instructions		
Breakpoint Trap	BPT	BPT
Flag Trap (Conditional)	FLAG	FLAG
Supervisor Call Trap	SVC	SVC
Trap/Interrupt Returns		
Return from Trap*	RETT	RETT
Return from Interrupt*	RETI	RETI

* Privileged instruction (see note).

Branches transfer control to an instruction nonsequentially. The JUMP instruction allows the destination address to be specified using a general choice of addressing modes. The BR instruction also transfers control, but provides a more code-compact form for PC-relative references. The Bcond instruction performs a branch as per the BR instruction if a specified condition code is true. The CASEi instruction branches by adding the contents of any general operand to the

Program Counter. In conjunction with Scaled Indexing (Section 4.4.9), this implements a multiway branch which corresponds directly to the Pascal CASE statement and the C SWITCH statement. The ACBi (Add, Compare and Branch) instruction supports looping by adding a small increment (range -8 to +7) to any general operand and branching if the result is non-zero.

Local procedure calls (JSR and BSR) transfer control as per the JUMP and BR instructions, respectively, except that they first save the address of the next sequential instruction onto the current stack as a 32-bit return address. The called procedure returns control after such a call with the RET instruction.

External procedure calls are implemented by the CXP and CXPD instructions. An external procedure is defined as a procedure which is in another module from the procedure currently executing. See Section 2.7.2 for further details of the module environment implemented by the Series 32000 architecture. An external procedure call saves the current contents of the MOD register as well as the return address onto the current stack, sets up the MOD and SB registers to match the environment of the destination module, and transfers control. In the CXP instruction, the destination procedure is specified with an index into the Link Table belonging to the current module, from which a descriptor is read, locating the destination. In the CXPD instruction, this descriptor is given as a general operand, greatly facilitating references to procedures which have themselves been passed as parameters. (A procedure can be passed as a parameter by passing its descriptor, using the LXPD form of the ADDR instruction.) The RXP instruction is used to return control after an external procedure call, restoring the MOD and SB registers as well as the Program Counter.

Three instructions have the function of causing deliberate traps. The BPT, FLAG and SVC instructions each have unique vectors in the Interrupt Dispatch Table (Section 2.7.4). The BPT instruction is intended to support debug breakpointing of programs. The FLAG instruction causes a trap if the PSR F bit is set (e.g. if the previous ADD instruction overflowed), and the SVC instruction provides the mechanism to make requests of a protected operating system.

The RETT instruction returns control from a trap or the Non-Maskable or Non-Vectored interrupt, restoring the PSR, MOD and SB registers. Since traps are often caused deliberately to request service of an operating system, the RETT instruction also allows parameters on the top of the original stack to be discarded in the process of returning. The RETI instruction is used for returning from any vectored maskable interrupt, providing the function of the RETT instruction and also communicating with one or more NS16202 Interrupt Control Units to implement transparent interrupt control.

NOTE: The instructions RETT and RETI are privileged, because they may change the contents of the high-order byte of the PSR, which is protected. The Illegal Operation trap, Trap(ILL), will occur if either of these instructions is attempted by a program in User Mode (i.e., while the PSR U bit is set).

3.11 Processor Service Instructions

Processor service instructions provide general housekeeping functions and services. The following is a list of the processor service instructions:

Instructions	Mnemonic Forms	Index
Effective Address		
Calculate Effective Address	ADDR	ADDR
Load External Procedure Descriptor (alternate mnemonic for ADDR)	LXPD	LXPD
Context Instructions		
Save General Purpose Registers	SAVE	SAVE
Restore General Purpose Registers	RESTORE	RESTORE
Enter New Procedure Context	ENTER	ENTER
Exit Procedure Context	EXIT	EXIT
Register/Stack Manipulation		
Adjust Stack Pointer	ADJSPB, ADJSPW, ADJSPD	ADJSPi
Bit Clear in PSR*	BICPSRB, BICPSRW	BICPSRB BICPSRW
Bit Set in PSR*	BISPSRB, BISPSRW	BISPSRB BISPSRW
Load Processor Register*	LPRB, LPRW, LPRD	LPRi
Store Processor Register*	SPRB, SPRW, SPRD	SPRi
Set Configuration Register*	SETCFG	SETCFG
Miscellaneous		
No Operation	NOP	NOP
Wait for Interrupt	WAIT	WAIT
Diagnose	DIA	DIA

* Privileged, or having privileged forms (see note).

There is one effective address instruction, ADDR, which calculates the effective address of its first operand and places that 32-bit address into its second operand location. The mnemonic LXPD (Load External Procedure Descriptor) is provided as a specific form of the ADDR instruction which reflects the action of ADDR when its first operand is specified using the External addressing mode (Section 4.4.6) and is an external procedure rather than external data. See the ADDR and LXPD instruction descriptions.

Context instructions allow the saving and restoring of portions of the processor context to and from the current stack. The SAVE instruction pushes the contents of any set of General-Purpose registers specified by the programmer. The RESTORE instruction undoes this by popping information from the top of the stack into any set of these registers. The ENTER and EXIT instructions deal with a larger context which is used by both local and external procedures. The ENTER instruction is generally the first instruction executed in a procedure, and has the function of completing the "activation record" or "stack frame". It saves the Frame Pointer (FP) register onto the current stack, allocates a specified number of bytes on the stack to be used for dynamic local variables, and sets up the Frame Pointer as a base pointer for this area. It also pushes the contents of any specified General-Purpose registers, as per the SAVE instruction. After executing this instruction, the Frame Pointer can be used in the Frame Memory and Frame Memory Relative addressing modes (Sections 4.4.8 and 4.4.3) to access both these local variables and any parameters passed to this procedure. The EXIT instruction is placed at the end of the procedure, undoing the action of the matching ENTER instruction. It restores the contents of the specified General-Purpose registers from the stack, discards the local variable space, and restores the Frame Pointer, leaving the return address at the top of the stack for the appropriate Return instruction.

Register/Stack Manipulation instructions provide the means to load, store and adjust the contents of CPU dedicated registers. (Corresponding instructions for manipulating dedicated Floating-Point and Memory Management registers are listed in Sections 3.3 and 3.12.) The ADJSPi instruction provides the means to directly adjust the current Stack Pointer register by the contents of any general operand in order to allocate or purge space on the stack or for alignment purposes. The BICPSR and BISPSR instructions allow specified bits in the PSR register to be cleared or set without affecting the rest of the PSR. The LPRi and SPRi instructions load or store a specified dedicated register. The SETCFG instruction sets up the CFG register (Section 2.3) to declare the presence of external interrupt control and slave processors.

Three instructions provide miscellaneous functions. The NOP (No Operation) instruction is a 1-byte instruction which does nothing except transfer control to the next sequential instruction. The WAIT instruction causes instruction processing to be suspended until an interrupt occurs. The DIA instruction provides a function similar to WAIT for hardware breakpointing purposes, but is not intended for use in programming.

NOTE: The instructions flagged with an asterisk ("*") have forms which are privileged. The Illegal Operation trap, Trap(ILL), will occur if they are attempted in User Mode (i.e., while the PSR U bit is set). The BICPSRW and BISPSRW instruction forms are privileged, as they may change the high-order byte of the PSR, which is protected. The LPRi and SPRi instructions are privileged when they reference either the INTBASE register or the entire PSR. The SETCFG instruction is privileged always.

3.12 Memory Management Instructions

The following is a list of the Memory Management instructions:

Instruction	Mnemonic Forms	Index
Load Memory Management Register	LMR	LMR
Store Memory Management Register	SMR	SMR
Validate Address for Reading	RDVAL	RDVAL
Validate Address for Writing	WRVAL	WRVAL
Move Value from Supervisor to User Space	MOVSUB, MOVSUW, MOVSUD	MOVSUi
Move Value from User to Supervisor Space	MOVUSB, MOVUSW, MOVUSD	MOVUSi

The LMR and SMR instructions load and store the contents of Memory Management registers (Section 2.5) as 32-bit values. The RDVAL instruction tests the protection level of a specified user memory location to determine whether the current user-mode program is allowed to read it. The WRVAL instruction tests whether the current user is allowed to write into a specified memory location. The MOVSUi instruction moves a byte, word, or double-word value from a specified location in the Supervisor addressing space to a location in the User space, and the MOVUSi instruction moves a value from User space to Supervisor space.

NOTES:

1. If the M bit in the CFG register has not been set (by the SETCFG instruction), the LMR, SMR, RDVAL and WRVAL instructions will generate the Undefined Instruction trap, Trap(UND).

2. All Memory Management instructions are privileged. If attempted by a program running in User Mode (i.e., while the PSR U bit is set), the Illegal Operation trap, Trap(ILL), will occur instead.

3.13 Custom Instructions

A set of instructions has been set aside for custom use. These instructions are reserved for such use, and will not be defined otherwise by NSC.

A custom instruction starts with one of the following binary encodings as its least-significant byte:

1. 00010110
2. 00110110
3. 10110110

Note that each of these corresponds to the first byte of a Floating-Point or Memory Management instruction, the difference being that bit 3 is "0" instead of "1".

If the C bit in the CFG register is cleared (by the SETCFG instruction), these instructions cause the Undefined Instruction trap, Trap(UND). Since a trap pushes the address of this first byte as the return address, the format and length of the remainder of the instruction may be defined in any manner, as required by the custom application.

If the C bit in the CFG register is set, these instructions are executed by an external "Custom" Slave Processor. The remainder of each instruction must follow the format of its corresponding Floating-Point or Memory Management instruction. The custom instructions corresponding to Memory Management instructions are privileged. In executing a custom instruction, the operand definitions and the protocol followed in communicating with the Custom Slave are identical to those for the corresponding Floating-Point or Memory Management instruction.

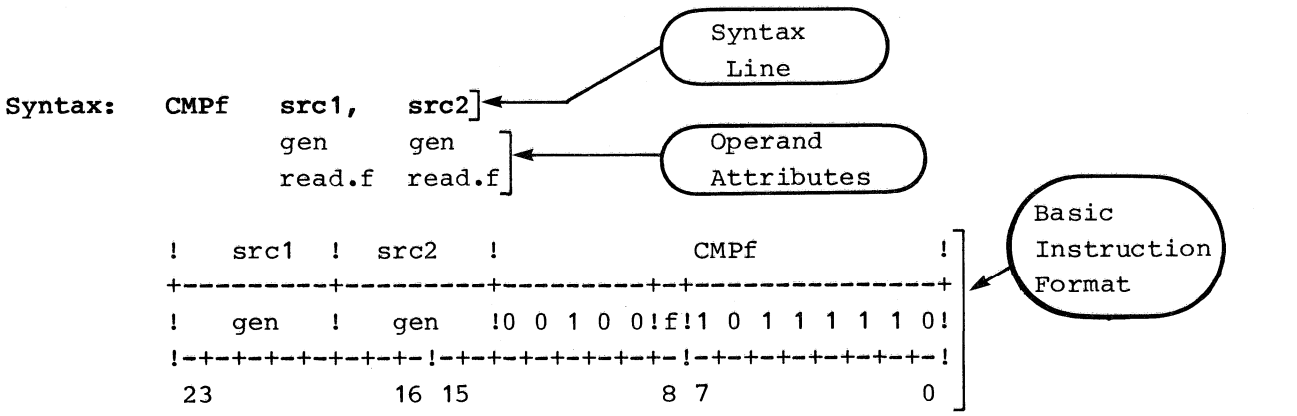
See the applicable CPU data sheet for details of the instruction formats and the Slave Processor protocols used.

Chapter 4

INSTRUCTION OPTIONS AND CONSTRUCTION

This chapter defines the options available in Series 32000 instructions, how these options are denoted in Chapter 5 (Instruction Set), and how the binary form of an instruction is constructed based on the selections made.

The structure of an instruction is given in Chapter 5 by its format definition. A typical format definition follows:

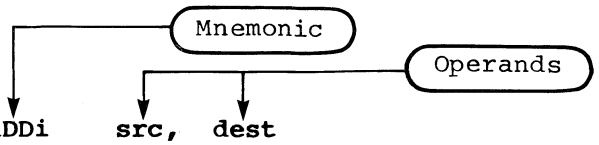


The notations used are defined in the following sections:

Other information presented in this chapter:

4.1 Syntax Presentation

The Syntax line presents the instruction mnemonic, followed by a list of operands, as shown. Lower-case items indicate options to be specified by the programmer.



Within the mnemonic, the following lower-case items may appear:

i An integer operation length suffix. It is specified by the programmer as

B = Byte (8-bit integer operation)
W = Word (16-bit integer operation)
D = Double-Word (32-bit integer operation)

and defines the length of the operation to be performed. In arithmetic operations, the carry and overflow tests use this specification to determine which bit positions are to be checked. When an implied operand of attribute "quick" appears (Section 4.2.3), it is internally sign-extended to this length before use. The lengths of integer general operands are usually taken from this length, but this depends on their individual length attributes, Section 4.2.2.

f A floating-point operation length suffix. It is specified by the programmer as

F = Single-precision Floating (32-bit floating-point operation)
L = Double-precision Long Floating (64-bit floating-point operation)

and defines the length of the operation performed. The lengths of floating-point general operands are usually taken from this length specification, but this depends on their individual length attributes, Section 4.2.2. In certain conversion instructions (e.g. ROUNDfi) both integer and floating-point operation lengths may appear.

cond A condition code, as in the Conditional Branch instruction:

Syntax: Bcond dest

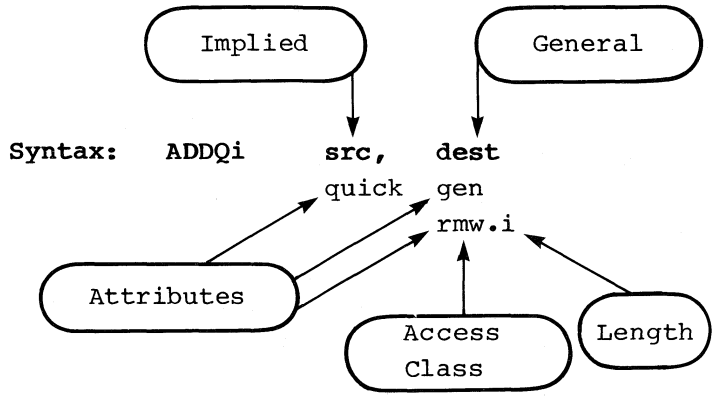
The specifications allowed and their interpretations are listed in the instruction description.

Operands are always given in lower case, and are to be specified by the programmer according to the attributes appearing below them (Section 4.2). The name given to an operand on the Syntax line serves to identify it in the instruction description.

4.2 Operand Attributes

Operands are defined in Chapter 5 by a set of attributes. These define what may be specified for each operand, and exactly how any valid operand specification will be interpreted when the instruction is executed.

A typical set of attributes is shown below:



Some operands listed as part of the instruction syntax are implied, meaning that their locations are not determined from a general choice of addressing modes. An implied operand is identified by the attribute "reg", "quick", "short", "imm" or "disp"; i.e., anything except "gen". For the explanations of implied operand attributes, see Section 4.2.3.

Most Series 32000 operands, however, are general, meaning that a general choice of addressing modes (Section 4.4) may be used to specify their locations. General operands are identified by the attribute "gen". A general operand has the additional attributes of an access class and also a length where relevant.

The access class attribute serves to define all cases of addressing mode usage including exceptional cases whose effects (or even legality) might not otherwise be obvious. The possible access classes for a general operand are read, write, rmw, addr and regaddr. Three addressing modes are affected by the access class: Register, Immediate and Top of Stack, as shown in Table 4-1 and described in Section 4.2.1.

The length attribute defines a general operand's data type and its size in bytes (see Section 4.2.2).

An operand with attribute i is an integer of the size given as the integer operation length by the programmer. An operand with attribute 2i is twice this size. An operand with attribute B, W or D is a byte, word or double-word integer, respectively, regardless of the operation length.

An operand with length attribute f is a floating-point value of the size given as the floating-point operation length by the programmer. An operand with length attribute F or L is a single-precision or double-precision floating-point value, respectively, regardless of the operation length.

4.2.1 Access Classes

Computer architectures usually have exceptional cases of operand reference based on the context of the instruction making the reference. For example, if an architecture allows references to registers as general operands, and provides a Jump instruction specifying a general destination, an obvious question becomes whether in this context (Jump) it is still legal to specify a register.

Rather than defining the interpretations of operand references on an instruction-by-instruction basis, the Series 32000 architecture defines five standard contexts (access classes) within which an Series 32000 family CPU will interpret a reference to a general operand. Each instruction assigns one access class to each of its general operands, which in turn fully defines the action of any addressing mode in referencing that operand.

Only three addressing modes have interpretations which are affected by the access class of an operand. These are Register, Immediate and Top of Stack. The five access classes, defined below, are read, write, rmw, addr and regaddr. See also Table 4-1.

read: The addressing modes are interpreted in the context of an operand being read but not rewritten. If Register mode is used, the specified register contains the operand. Immediate mode is legal only for operands of this access class. If Top of Stack mode is specified, the Stack Pointer is post-incremented by the number of bytes corresponding to the length of the operand (as determined from its length attribute, Section 4.2.2), thus "popping" it from the stack.

write: The addressing modes are interpreted in the context of an operand being written without having been read. If Register mode is used, the specified register receives the operand. Immediate mode is undefined for this access class. If Top of Stack mode is specified, the Stack Pointer is pre-decremented by the number of bytes corresponding to the length of the operand (as determined from its length attribute, Section 4.2.2), thus "pushing" it onto the stack.

rmw: Read-Modify-Write. The addressing modes are interpreted in the context of an operand being read, modified and rewritten to the same location. If Register mode is used, the specified register contains the operand. Immediate mode is undefined for this access class. If Top of Stack mode is specified, the Stack Pointer provides the address of the operand, but is not altered.

Table 4-1 Addressing Mode Actions vs. Access Class

Addressing Mode	Access Class				
	read	write	rmw	addr	regaddr
Register	Rn, Fn	Rn, Fn	Rn, Fn	(Rn)	Rn, Fn
Immediate	legal	undefined	undefined	undefined	undefined
Top of Stack	Pop	Push	(SP)	(SP)	(SP)

NOTES: 1. The notations (Rn) and (SP) signify use of the enclosed register as a pointer. The register is not altered.

2. Using Scaled Indexing in an addressing mode overrides the access class and forces it to "addr".

addr: Address. The addressing modes are interpreted in the context of an operand which cannot be held in a register, or of an effective address calculation which does not correspond to an operand being fetched as data. Examples of this context are ADDR A,B (place the effective address of A into B), JUMP X (place the effective address of X into the Program Counter) or any addressing mode using Scaled Indexing (since arrays cannot be held in registers; see Table 4-1). If Register mode is used, the operand is in memory, and the specified register contains its address. Immediate mode is undefined for this access class. If Top of Stack mode is specified, the Stack Pointer provides the address of the operand, but is not altered.

Note: The addr access class does not define the use to which an operand is put, but only the context in which the addressing modes are interpreted. An addr operand may be read, written, or neither read nor written, depending on the instruction being executed.

regaddr: Register/Address. The addressing modes are interpreted in the context of designating a base for locating a data item of nonstandard size and/or alignment. An example of this context is the operand B in the instruction TBITW A,B (test the bit which is A bits from the beginning of base location B). If Register mode is used, the data item is held within the specified register. Immediate mode is undefined for this access class. If Top of Stack mode is specified, the Stack Pointer provides the address of the base, but is not altered.

Note: The regaddr access class does not define the use to which an operand is put, but only defines the context in which the addressing modes are interpreted. Information at the location given in a regaddr context may be read, written, or neither read nor written, depending on the instruction being executed.

4.2.2 Length Attributes

The length attribute of a general operand defines its data type and its length (in bytes). Operands with length attribute B, W, D, i or 2i are integers. Operands with length attribute F, L or f are floating-point values.

The length in bytes of an operand affects the following three addressing modes:

Register: If the length of an operand is smaller than the designated General-Purpose register, it is only the low-order portion of the register which is referenced or modified. The rest of the register is unchanged. Operands with length attribute 2i are a special case; see Section 4.2.2.1 below.

Immediate: The length of the value held within the binary instruction format matches the length in bytes of the operand.

Top of Stack: If the access class attribute (Section 4.2.1) indicates that the Stack Pointer is to be modified, it is modified by the operand length in bytes.

4.2.2.1 Integer Length Attributes

The length attributes which identify an integer are B, W, D, i and 2i. For integers, the Register addressing mode assumes that the General-Purpose registers (R0-R7) are to be used. Floating-Point registers cannot be specified for integer operands. The integer length attributes are defined as follows:

B The operand is a 1-byte integer.

W The operand is a 2-byte (word) integer.

D The operand is a 4-byte (double-word) integer.

i The operand is either one, two, or four bytes in length, depending on the operation length suffix (B, W or D: Section 4.1) appended to the instruction mnemonic by the programmer.

2i The operand is twice the length given as the operation length suffix (Section 4.1) appended to the instruction mnemonic by the programmer.

The MEI and DEI instructions (Multiply/Divide Extended Integer) present special cases in which operands with length attribute 2i can be held in registers. If an operand with length attribute 2i is specified as being within a register, it occupies a pair of General-Purpose registers (R0 and R1, R2 and R3, R4 and R5, or R6 and R7), and the even-numbered register of the pair must be specified as the operand location. The operand is held with its least-significant half in the even-numbered register (right-justified) and its most-significant half in the odd-numbered register (also right-justified). Any portions of the two registers not used to hold the operand are neither referenced nor modified.

4.2.2.2 Floating-Point Length Attributes

The length attributes which identify a floating-point operand are F, L and f. For floating-point operands the Register addressing mode assumes that the Floating-Point registers (F0-F7) are to be used. General-Purpose registers cannot be specified for floating-point operands. The floating-point length attributes are defined as follows:

F The operand is a 4-byte single-precision floating-point value.

L The operand is an 8-byte double-precision ("Long") floating-point value. If the Register addressing mode is specified for an operand of this length, then a pair of registers (F0 and F1, F2 and F3, F4 and F5, or F6 and F7) holds the operand, and only an even-numbered register may be specified. The low-order half of the operand is then held in the specified even-numbered register, and the high-order half is held in the odd-numbered register.

f The operand is either a single-precision or double-precision floating-point value, depending on the operation length suffix (F or L, Section 4.1) appended to the instruction mnemonic by the programmer. See the description of "L" above for the format of a double-precision operand within registers.

4.2.3 Implied Operand Attributes

Implied operands are specified without using addressing modes. Their attributes define how they may be specified.

reg: The operand location is a General-Purpose register (R0-R7). Any General-Purpose register may be specified. The entire register is always used and/or modified by the instruction. The register number is encoded in the binary instruction format within a 3-bit field marked "reg".

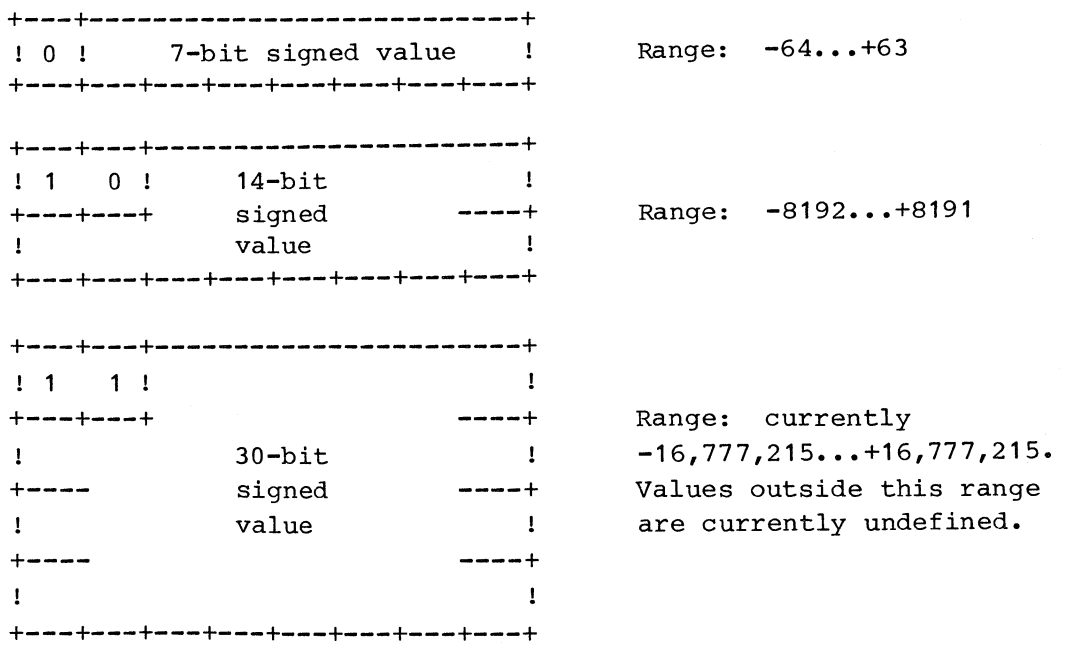
quick: The operand is a signed, 4-bit immediate value. Its range is -8 to +7. Before use, it is internally sign-extended to the length given by the operation length suffix appended to the instruction mnemonic. A quick operand is encoded in the binary instruction format within a 4-bit field marked "quick".

short: The operand occupies a 4-bit field within the binary instruction format. The interpretation of the field depends on the instruction.

imm: The operand is a 1-byte immediate value, appended to the instruction following any addressing extensions. Its interpretation is determined by the instruction.

disp: The operand is an immediate signed integer value, encoded as a displacement field and appended to the instruction following any addressing extensions. Its use is determined by the instruction.

A displacement field is stored with the most-significant byte at the lowest address. Its format is determined by its most-significant bits as shown below.



4.3 Binary Instruction Format

The binary format of an Series 32000 instruction is shown in Figure 4-1. It is divided into two sections.

1. The Basic Instruction portion defines the operation performed and the number and kinds of operands. It is presented in Chapter 5 individually for each instruction, using field nomenclature as defined in Section 4.3.1 below.

2. Extension fields are optionally appended as defined by the instruction and the addressing modes chosen by the programmer. These extensions fall into a general instruction format, defined in Section 4.3.2.

Because the Series 32000 family implements a full two-address architecture, most instructions have two general operands (with attribute "gen", Section 4.2). To distinguish between them, the first general operand appearing in the Syntax line of an instruction description will be designated Operand A and the second Operand B.

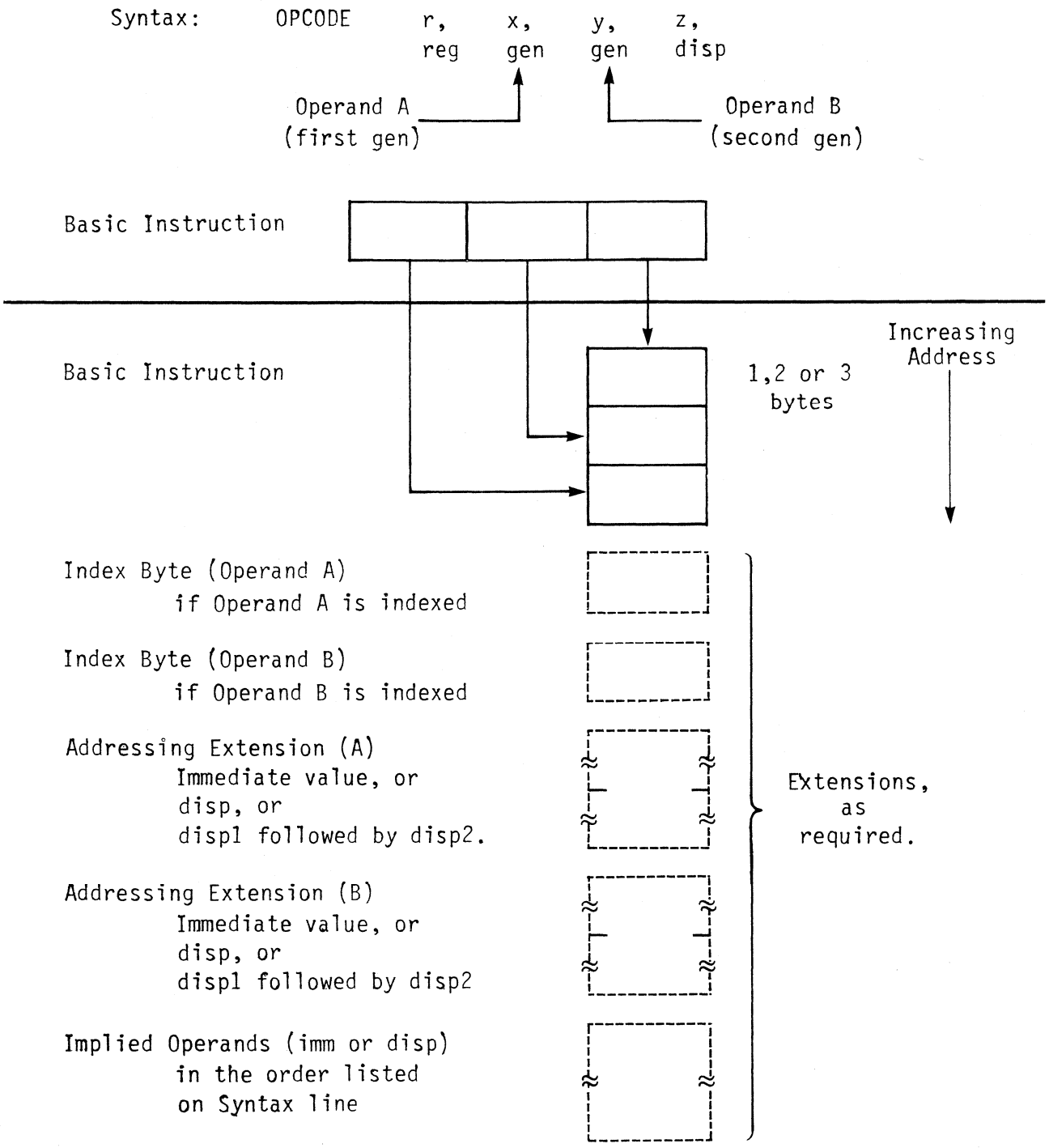


Figure 4-1 General Format EZ-01-0

4.3.1 Basic Instruction

The Basic Instruction portion defines the operation performed and the addressing modes used for referencing general operands, and provides fields within it for holding all implied operands with attribute reg, quick or short (Section 4.2.3). It is one, two or three bytes in length.

The format of the Basic Instruction is diagrammed for each instruction under the Syntax line of the instruction description. The format used for storing the Basic Instruction in memory is the same as for data elements; that is, the least-significant byte appears first, at the lowest address. Fields within the Basic Instruction are presented as defined below.

4.3.1.1 Operation Code Fields

Operation code fields are presented explicitly in binary. All fields presented in this manner are derived from the instruction mnemonic and define the basic operation to be performed.

4.3.1.2 Operation Length Fields: i and f

Operation Length fields define the length to which calculations are performed within a basic data type (integer or floating point). They also define the lengths of most general operands (indirectly, through each operand's own length attribute, Section 4.2.2). They are derived from the Operation Length mnemonic suffix (Section 4.1) chosen by the programmer, as shown below.

Field	Mnemonic Suffix	Encoding
i	B	00
	W	01
	D	11
f	F	1
	L	0

4.3.1.3 General Addressing Mode Fields: gen

These are 5-bit fields which define the addressing mode used to access each operand. The name of the operand from the Syntax line appears above the field. The encodings of these fields are given in the definitions of the addressing modes, Section 4.4.

4.3.1.4 Implied Operand Fields: reg, quick, short

These fields hold the necessary information for implied operands which are defined with the corresponding attribute (reg, quick or short; Section 4.2.3). The name of the operand from the Syntax line appears above the field.

A reg field is a 3-bit field holding a register number (0-7).

A quick field is a 4-bit field holding a signed value (range -8 to +7).

A short field is a 4-bit field holding information which is required by the individual instruction. Its contents are defined in the instruction description.

4.3.2 Extension Fields

The following fields extend the length of the instruction beyond the Basic Instruction field. They appear as required by the individual instruction or by the addressing modes chosen for specifying its general operands.

4.3.2.1 Index Bytes

The first form of extension is in the form of Index Bytes. The instruction is extended in this manner whenever Scaled Indexing (Section 4.4.9) is used in specifying a general operand. Either or both of the general operands may be specified using Scaled Indexing. If both operands are specified in this form, then the Index Byte for Operand A appears before the Index Byte for Operand B. See Figure 4-1. The format of an Index Byte is given in the definition of Scaled Indexing, Section 4.4.9.

4.3.2.2 Addressing Extensions

An addressing extension is appended for each general operand as required. Its contents depend on the addressing mode chosen for each. See Section 4.4 for the usages of addressing extensions in addressing modes. The addressing extension for operand A appears before the one for operand B (Figure 4-1).

Addressing extensions are constructed from two basic elements: displacement fields and immediate values.

NOTE: Unlike other values in memory, addressing extensions are ordered with the most-significant byte at the lowest address.

An addressing extension contains either:

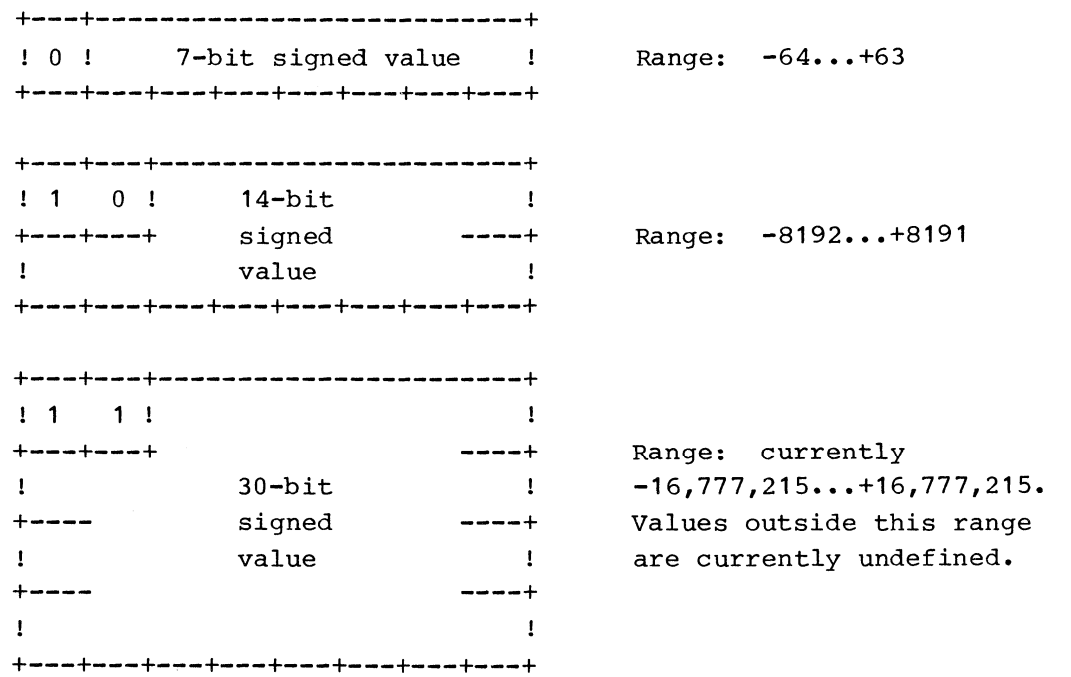
1. One immediate value, or

2. One displacement field, labelled "disp" in the addressing mode definitions (Section 4.4), or

3. Two displacement fields, labelled "disp1" and "disp2". In this form, disp1 is appended first, followed by disp2.

If a Register or Top of Stack addressing mode is used to specify a general operand, no addressing extension appears for that operand.

A displacement field holds a signed two's-complement addressing constant. It is stored with the most-significant byte at the lowest address. Its length is determined by its most-significant bits as shown below.



An immediate value appears as an addressing extension only when the Immediate addressing mode is specified (Section 4.4.4). The length of the value is determined from the operand's length attribute (Section 4.2.2). The value is ordered with its most-significant byte at the lowest address.

4.3.2.3 Implied Operand Extensions: imm, disp

Implied operands, of attribute "imm" or "disp" (Section 4.2.3), appear last, after all addressing extensions. If there is more than one imm or disp operand appearing in the instruction, then the operands are appended in the order in which they are listed on the Syntax line.

4.4 Series 32000 Addressing Modes

Any general operand (Section 4.2) may be specified by the programmer using a general choice of addressing modes. This section defines addressing mode syntax, functions and encodings.

Table 4-2 lists the addressing modes provided for specifying a general operand. It also serves as an index to this section. The Encoding column gives the binary encoding used in a gen field (Section 4.3.1.3) to select each mode. The Name column gives the name of the addressing mode as used in this manual, and the Syntax column shows the syntax used in assembly language to express it. (Note: What is given is only the lowest level of expression, which most directly relates to the action of the addressing mode. See the applicable assembler manual for full details of expression syntax and symbolic features.)

Scaled Indexing is an option available as part of any addressing mode except Immediate. It does not stand alone as an addressing mode, but is listed with the addressing modes because of the binary encodings used to select the option.

Table 4-2 Series 32000 Addressing Modes

	Encoding	Name	Syntax
Register	00000	Register 0	R0 or F0
	00001	Register 1	R1 or F1
	00010	Register 2	R2 or F2
	00011	Register 3	R3 or F3
	00100	Register 4	R4 or F4
	00101	Register 5	R5 or F5
	00110	Register 6	R6 or F6
	00111	Register 7	R7 or F7
Register Relative	01000	Register 0 Relative	disp(R0)
	01001	Register 1 Relative	disp(R1)
	01010	Register 2 Relative	disp(R2)
	01011	Register 3 Relative	disp(R3)
	01100	Register 4 Relative	disp(R4)
	01101	Register 5 Relative	disp(R5)
	01110	Register 6 Relative	disp(R6)
	01111	Register 7 Relative	disp(R7)
Memory Relative	10000	Frame Memory Relative	disp2(disp1(FP))
	10001	Stack Memory Relative	disp2(disp1(SP))
	10010	Static Memory Relative	disp2(disp1(SB))
(reserved)	10011	(Reserved for future use.)	
Immediate	10100	Immediate	value
Absolute	10101	Absolute	@disp
External	10110	External	EXT(disp1)+disp2
Top of Stack	10111	Top of Stack	TOS
Memory Space	11000	Frame Memory	disp(FP)
	11001	Stack Memory	disp(SP)
	11010	Static Memory	disp(SB)
	11011	Program Memory	* + disp
Scaled Indexing	11100	Byte Indexed	basemode[Rn:B]
	11101	Word Indexed	basemode[Rn:W]
	11110	Double-Word Indexed	basemode[Rn:D]
	11111	Quad-Word Indexed	basemode[Rn:Q]

4.4.1 Register Modes

Mode	Syntax	Encoding
Register 0	R0 or F0	00000
Register 1	R1 or F1	00001
Register 2	R2 or F2	00010
Register 3	R3 or F3	00011
Register 4	R4 or F4	00100
Register 5	R5 or F5	00101
Register 6	R6 or F6	00110
Register 7	R7 or F7	00111

Extensions

None.

The interpretation of these modes is formally defined below. However, rule 6 defines the general case, which is that the specified General-Purpose register (R0-R7) holds the operand.

The following rules are listed in order of decreasing precedence. Lower-numbered rules take precedence over higher-numbered rules.

1. If the access class of the operand (Section 4.2.1) is "addr", then the operand is in memory. The effective address of the operand is held in the specified General-Purpose register.

2. If Scaled Indexing is used, the access class of the operand is redefined as "addr", and rule 1 above applies.

3. If the operand length attribute (Section 4.2.2) is "2i", then a pair of General-Purpose registers (R0 and R1, R2 and R3, R4 and R5, or R6 and R7) holds the operand. The even-numbered register of the pair must be specified, and if the odd-numbered register is specified the location of the operand is undefined. The least-significant half of the operand is held in the low-order portion of the even-numbered register, and the remaining portion of the register is neither used nor affected. The most-significant half of the operand is held in the low-order portion of the odd-numbered register, and any remaining portion of the register is neither used nor affected.

4. If the operand length derived from its length attribute (Section 4.2.2) is single-precision floating-point, then the operand is held in the specified Floating-Point register (F0-F7).

5. If the operand length derived from its length attribute (Section 4.2.2) is double-precision floating-point, then the operand is held in a pair of Floating-Point registers (F0 and F1, F2 and F3, F4 and F5, or F6 and F7). The even-numbered register of the pair must be specified, and if the odd-numbered register is specified the operand location is undefined. The least-significant half of the operand is held in the even-numbered register and the most-significant half is held in the odd-numbered register.

6. When none of the above exceptions apply, the operand is an integer held within the specified General-Purpose register (R0-R7). If the operand length derived from its length attribute is shorter than the full 32-bit length of the register, then the operand occupies the low-order portion of the register, and the remaining portion of the register is neither used nor affected.

4.4.2 Register Relative Modes

Mode	Syntax	Encoding
Register 0 Relative	disp(R0)	01000
Register 1 Relative	disp(R1)	01001
Register 2 Relative	disp(R2)	01010
Register 3 Relative	disp(R3)	01011
Register 4 Relative	disp(R4)	01100
Register 5 Relative	disp(R5)	01101
Register 6 Relative	disp(R6)	01110
Register 7 Relative	disp(R7)	01111

Extensions

One displacement field: disp.

The operand is in memory. Its effective address is the sum of the 32-bit contents of the specified General-Purpose register (R0-R7) and the displacement value sign-extended to 32 bits.

4.4.3 Memory Relative Modes

Mode	Syntax	Encoding
Frame Memory Relative	disp2(disp1(FP))	10000
Stack Memory Relative	disp2(disp1(SP))	10001
Static Memory Relative	disp2(disp1(SB))	10010

Extensions

Two displacement fields: disp1 followed by disp2.

The operand is in memory, at the address given by the sum of disp2 (sign-extended to 32 bits) and a 32-bit pointer in memory. The address of this pointer is generated by adding disp1 (sign-extended to 32 bits) and the contents of the specified register (FP, SP or SB). The symbol "SP" means the stack pointer which is currently selected by the S bit in the PSR (Section 2.2).

NOTE: The Stack Memory Relative mode uses the contents of the selected stack pointer as it was at the beginning of the instruction. The effective address is therefore independent of any changes made to the stack pointer by any Top of Stack mode appearing in the same instruction.

4.4.4 Immediate Mode

Mode	Syntax	Encoding
Immediate	value	10100

Extensions

The value, placed most-significant byte first.

The operand value is input from the addressing extension portion of the instruction. The value appears most-significant byte first. Its length in bytes is determined from the operand length attribute (Section 4.2.2). Floating-point as well as integer instructions may use Immediate mode.

NOTES:

1. Immediate mode is legal only for operands of access class "read". Any other use is undefined.

2. Immediate mode may not be used as the base mode for Scaled Indexing.

4.4.5 Absolute Mode

Mode	Syntax	Encoding
Absolute	@address	10101

Extensions

One displacement field: address.

The absolute address is specified. This address is encoded in the binary instruction as a displacement field of any length required to hold the address.

NOTE: Negative addresses are undefined.

4.4.6 External Mode

Mode	Syntax	Encoding
External	EXT(disp1)+disp2 or EXT(disp1)	10110

Extensions

Two displacement fields: disp1 followed by disp2. If disp2 is omitted in assembly language, it must still be included as a disp2 field containing zero.

The External addressing mode provides the means for a software module to access data within a data space outside of that module. The operand is referenced through the Link Table of the current module (Section 2.7.3). The value disp1 is a Link Table entry number, and disp2 is a final displacement added to the address provided from that Link Table entry.

The operand is in memory, at the address given by the sum of disp2 (sign-extended to 32 bits) and a 32-bit pointer in the current Link Table. The address of this pointer is generated by adding disp1, multiplied by four, and the contents of the 32-bit value at memory address MOD + 4. "MOD" is the contents of the MOD register, interpreted as a 16-bit unsigned number.

4.4.7 Top of Stack Mode

Mode	Syntax	Encoding
Top of Stack	TOS	10111

Extensions

None.

The operand is in memory, at the top of the current stack. It is pushed, popped, or neither pushed nor popped, as appropriate to the usage of the operand.

The stack pointer used is the stack pointer that is currently selected by the S bit in the PSR (Section 2.2).

The stack pointer is used by Top of Stack mode according to the access class of the operand. The rules below are listed in order of decreasing precedence. Lower-numbered rules take precedence over higher-numbered rules.

1. If the operand is of access class "rmw", "addr" or "regaddr", then the effective address of the operand is given by the contents of the stack pointer, and no increment or decrement is performed.

2. If Scaled Indexing is used, the access class of the operand is redefined as "addr", and rule 1 above applies.

3. If the operand is of access class "read", the operand is read from the address given by the contents of the stack pointer. The stack pointer is then incremented by the length in bytes of the operand, as determined from its length attribute (Section 4.2.2).

4. If the operand is of access class "write", the stack pointer is decremented by the length in bytes of the operand, as determined from its length attribute (Section 4.2.2). The operand is then written to the address given by the new contents of the stack pointer.

NOTES:

1. If Top of Stack mode is used for both general operands of an instruction, the operands are accessed and the stack pointer modified in left-to-right operand order. The rightmost addressing mode uses as its initial stack pointer value the contents of the stack pointer after any increment or decrement has been performed by the leftmost addressing mode.

2. The Stack Memory and Stack Memory Relative modes use as their stack pointer value the contents of the selected stack pointer as they were at the beginning of the instruction. The actions of these modes are therefore independent of any modifications made to the stack pointer by any Top of Stack mode appearing within the same instruction.

4.4.8. Memory Space Modes

Mode	Syntax	Encoding
Frame Memory	disp(FP)	11000
Stack Memory	disp(SP)	11001
Static Memory	disp(SB)	11010
Program Memory	* + disp	11011

Extensions

One displacement field: disp.

The operand is in memory, at the address given by the sum of the contents of the specified register and the displacement value sign-extended to 32 bits.

The symbol "SP" means the stack pointer (SP0 or SP1) which is currently selected by the S bit in the PSR (Section 2.2). The symbol "*" means the contents of the Program Counter.

NOTES:

1. The Stack Memory mode uses the contents of the selected stack pointer as it was at the beginning of the instruction. The effective address is therefore independent of any changes to the stack pointer contents made by any Top of Stack mode occurring in the same instruction.

2. The Program Counter always contains the address of the first byte of the instruction being executed.

4.4.9 Scaled Indexing

Mode	Syntax	Encoding
Byte Indexed	basemode[Rn:B]	11100
Word Indexed	basemode[Rn:W]	11101
Double-Word Indexed	basemode[Rn:D]	11110
Quad-Word Indexed	basemode[Rn:Q]	11111

basemode = base addressing mode (see below)

Rn = any General-Purpose Register, used as the index register.

Extensions

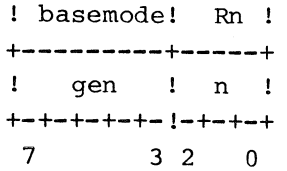
1. Index Byte.
2. Any extensions required by basemode.

Any addressing mode except Immediate is allowed to include indexing by the contents of any General-Purpose register (R0-R7), interpreted as a signed 32-bit integer. The index value is scaled (multiplied) by a factor of 1, 2, 4 or 8 before use, so that it can be used as an element number for an array of 1-, 2-, 4- or 8-byte elements. An indexed addressing expression has the form

basemode[Rn:l]

where basemode is an addressing mode expression,
Rn is any General-Purpose register, and
l is an element length qualifier, chosen from:
B = Byte, scale factor = 1
W = Word, scale factor = 2
D = Double-word, scale factor = 4
Q = Quad-word, scale factor = 8 .

In the binary instruction format, addressing modes with Scaled Indexing are encoded within the Basic Instruction gen field as one of four special codes which specify only the length qualifier (see table above). The basemode and Rn components are specified in an Index Byte appended to the Basic Instruction. See Section 4.3 for the position of an Index Byte in the general instruction format. The Index Byte has the following format:



Any further addressing extensions required by basemode are appended as given in Section 4.3.2.2, in exactly the same manner as if basemode were not indexed.

NOTES:

1. Any operand specified using Scaled Indexing is redefined as being of access class "addr" regardless of the operand's access class in the instruction definition. This affects the interpretation of basemodes Register and Top of Stack, and makes the use of an Immediate basemode illegal. See Section 4.2.1.

2. Scaled Indexing may be applied only once in an addressing expression. Basemode is therefore not allowed to include Scaled Indexing within itself.

4.5 Constructing Complete Binary Instructions: Some Examples

The following examples illustrate the process of assembling the binary form of an Series 32000 instruction from its assembly-language form.

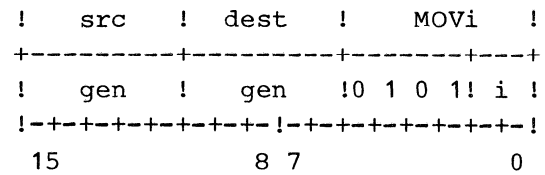
Example 1:

The simple example below is generated from the Move instruction (MOVi).

```
MOVB  R0, R1
```

This instruction copies the low-order byte of register R0 to the low-order byte of register R1. The format definition of the MOVi instruction is taken from Chapter 5 as shown below.

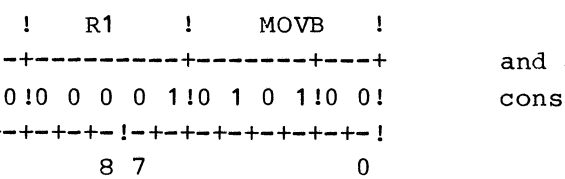
Syntax: **MOVi** **src,** **dest** MOVB
gen gen MOVW
read.i write.i MOVD



In this example, the lower-case items in the Syntax line have been specified by the programmer as follows:

```
i    =    B    (Byte operation length, Section 4.1)
src  =    R0   (Register 0 addressing mode, Section 4.4.1)
dest =    R1   (Register 1 addressing mode, Section 4.4.1)
```

To complete the Basic Instruction, the gen fields for the two general operands src and dest and the i field for the operation length must be provided. The encoding for the src operand (R0 Register addressing mode) is 00000. The encoding for the dest operand (R1 Register addressing mode) is 00001. The encoding for the operation length (B) is 00. Thus, the Basic Instruction is:



and appears in memory as the two consecutive bytes: 54 00 (Hex).

The Register addressing modes R0 and R1 require no addressing extensions. Therefore, the Basic Instruction above is the complete binary form of the example instruction.

Example 2:

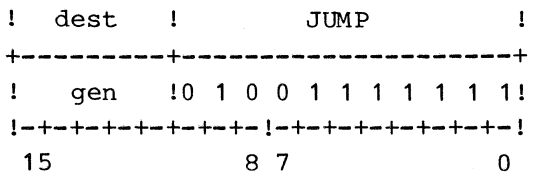
The next example is generated from the JUMP instruction.

```
JUMP  0(4(SB))
```

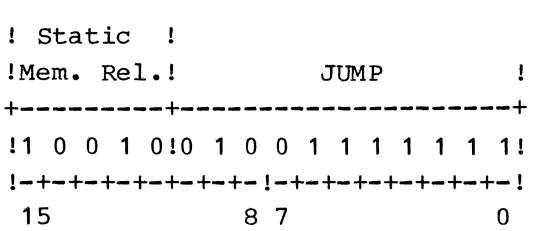
This instruction performs an indirect jump through a 32-bit pointer in memory. The pointer's address is calculated by adding 4 to the contents of the SB register.

The format definition of the JUMP instruction is:

Syntax: **JUMP** **dest**
gen
addr



This instruction has only one operand, the general operand dest, which is specified by the programmer with the addressing expression 0(4(SB)). This form of addressing expression specifies that the Static Memory Relative addressing mode (Section 4.4.3) is to be used to calculate the address to which the instruction will jump. The code for this addressing mode is placed in the gen field as binary 10010. Thus, the Basic Instruction is:



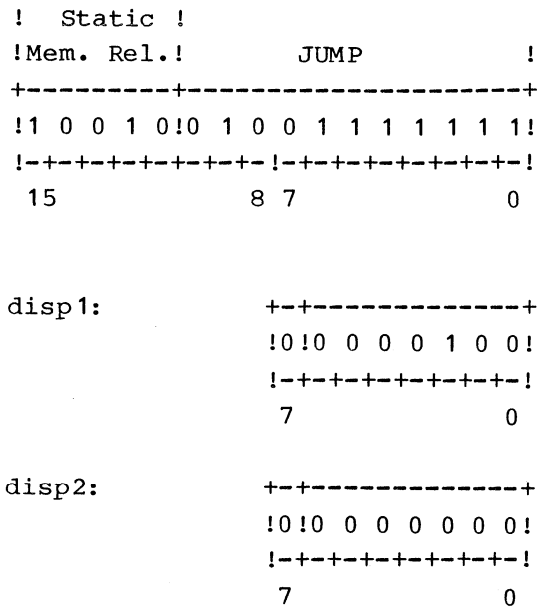
The Memory Relative addressing modes require that two displacements be appended to the Basic Instruction. These are designated disp1 and disp2. From the expression provided in the assembly-language example, the displacement values are to be:

disp1 = 4, and
disp2 = 0 .

(continued)

From the format given for a displacement field (Section 4.3.2.2), we see that a small value can be represented in either one, two or four bytes. Obviously, we wish to choose the smallest field which works, so we will use the 1-byte format for each displacement field.

Appending the two displacements to the Basic Instruction, we get the complete binary instruction as shown below.



The complete binary instruction is represented in consecutive memory bytes as

7F 92 04 00 (Hex).

Example 3:

The following example is generated from the ADDi instruction.

```
ADDD  EXT(8)+80, -4(FP)
```

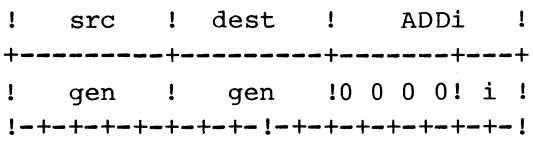
This instruction adds a 32-bit value from the memory location specified as EXT(8)+80 to a 32-bit value at the memory location specified as -4(FP).

The format definition of the ADDi instruction is:

Syntax: **ADDi** **src,** **dest** ADDB

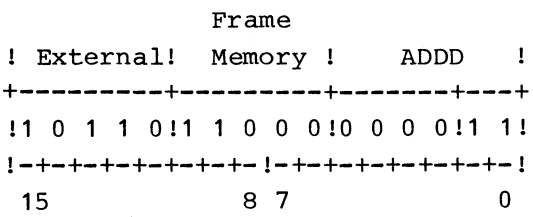
gen gen ADDW

read.i rmw.i ADDD



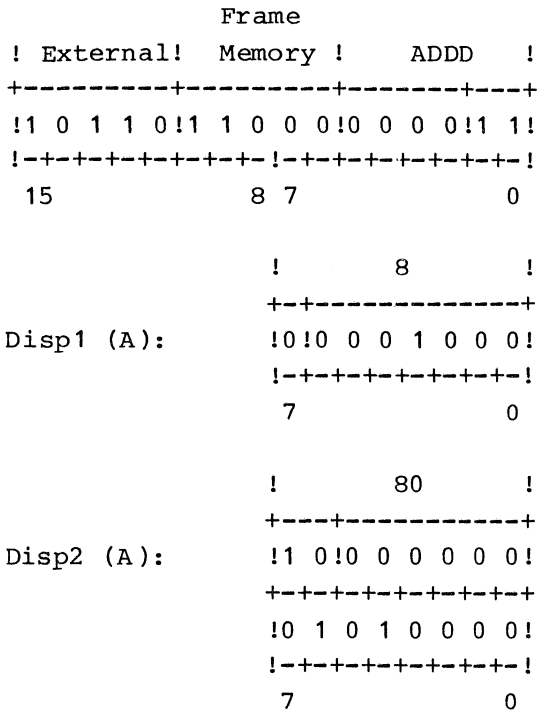
This instruction has two general operands. For purposes of constructing its binary form, the src operand is labeled operand A and the dest operand is labeled operand B, as discussed in Section 4.3.

The operation length suffix is D, encoded as 11 in the i field. The src operand is specified using the External addressing mode (Section 4.4.6), which is encoded in the binary instruction as 10110 in the corresponding gen field. The dest operand is specified using the Frame Memory addressing mode (Section 4.4.8), which is encoded in the corresponding gen field as 11000. The Basic Instruction appears then as shown below.



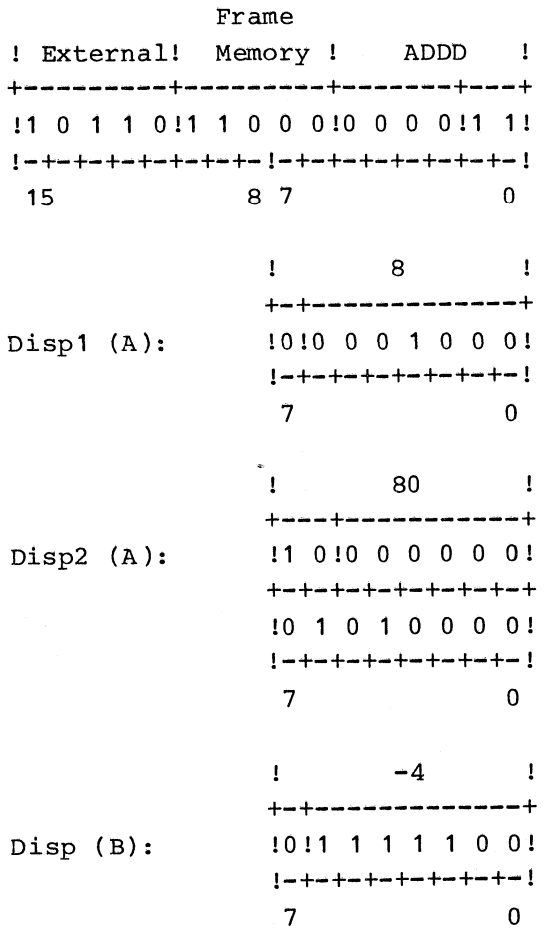
(continued)

Since neither operand uses Scaled Indexing, the first extensions appended to the Basic Instruction are the addressing extension fields required by the External addressing mode used to specify the src operand (Operand A). The External addressing mode requires two displacement fields: disp1 (containing 8) followed by disp2 (containing 80). The disp1 displacement value can be held in a single-byte displacement field. The disp2 displacement value cannot, as it is outside the range (-64 to +63) which can be represented in a signed 7-bit number. It can, however, be held in a two-byte displacement field. Appending the displacement fields for Operand A yields the result shown below.



(continued)

After the addressing extensions required for Operand A, the addressing extensions required for Operand B are appended. Since Operand B (the dest operand) is specified using the Frame Memory addressing mode, there is one displacement field required, containing the value -4. This value is within the range -64 to +63, and so it can be held in the single-byte displacement format. It is appended as shown:



The complete instruction appears in consecutive memory bytes as:

03 B6 08 80 50 7C (Hex).

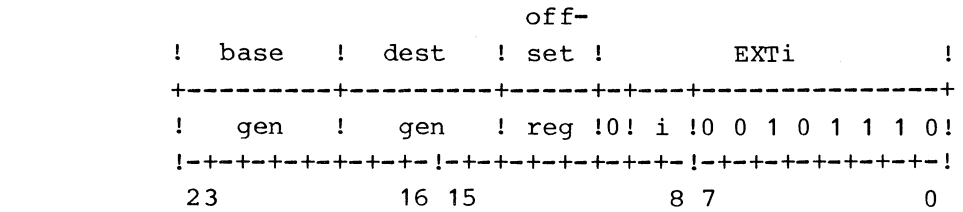
Example 4:

A final example of how an instruction is assembled uses the Extract Field (EXTi) instruction.

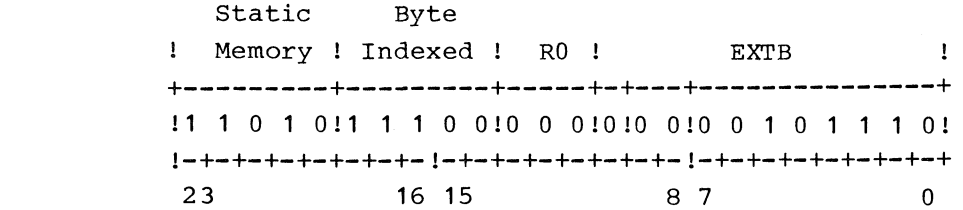
```
EXTB   R0, 10(SB), 0(SB)[R1:B], 5
```

This instruction copies a 5-bit field from a point in memory determined by a bit offset (contained in R0) from the address 10(SB) to the address specified by 0(SB)[R1:B]. The format definition of the Basic Instruction is:

Syntax: **EXTi** **offset,** **base,** **dest,** **length** EXTB
reg gen gen disp EXTW
regaddr write.i EXTD

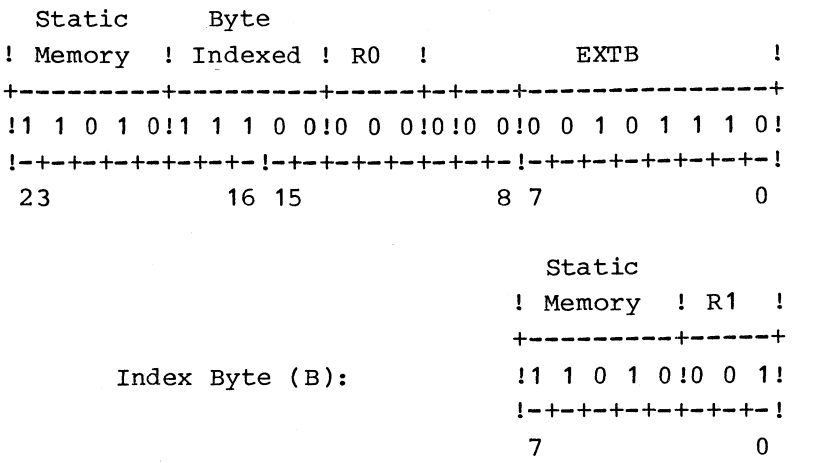


In this more complex instruction, there are several items which must be placed in the Basic Instruction. These are the addressing modes specified by the expressions 10(SB) and 0(SB)[R1:B], the i field corresponding to the B operation length suffix, and the reg field corresponding to the reg operand specified as R0. The code for the expression 10(SB), specifying the Static Memory addressing mode, is 11010. The code for the expression 0(SB)[R1:B], specifying the Static Memory addressing mode with Scaled Indexing (scale factor = 1), is 11100. (Note that when Scaled Indexing is used, it is the code for Scaled Indexing which is placed in the Basic Instruction. See Section 4.4.9.) The i field is 00, for the B operation length suffix. The reg field is 000, for R0. Thus, the Basic Instruction is:



(continued)

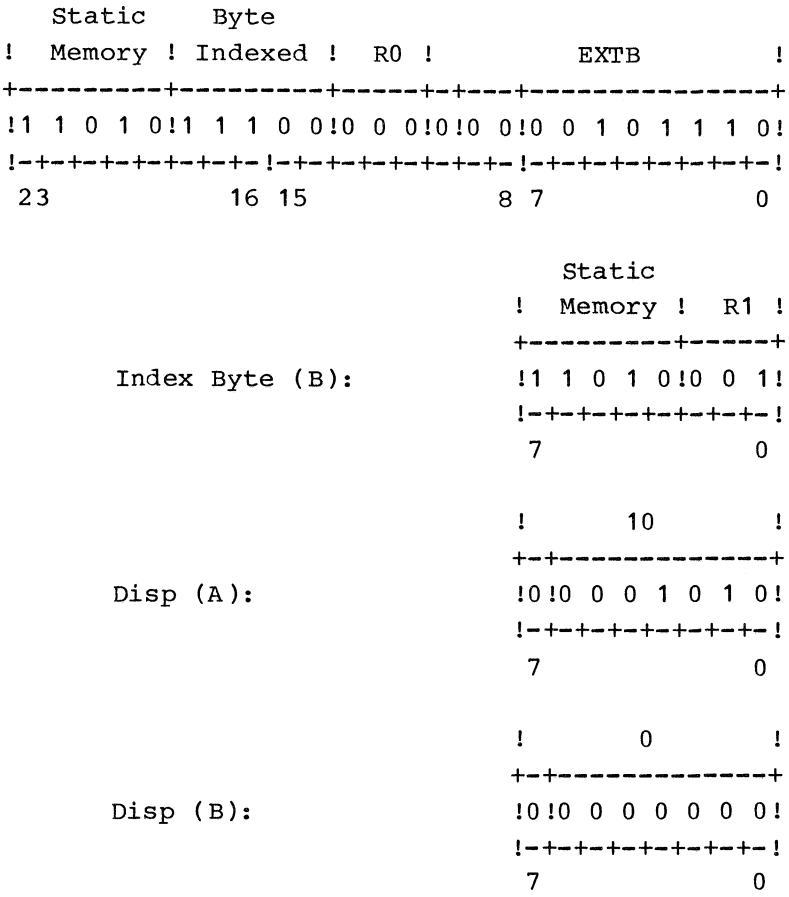
The expression 10(SB) specifies Operand A and 0(SB)[R1:B] specifies Operand B. Because it is indexed, Operand B requires an Index Byte. The Index Byte is the first extension to be appended to the Basic Instruction. It contains the code for the basemode 0(SB) and the register number for R1. The basemode (Static Memory) is encoded as 11010 and the register number is encoded for R1 as 001.



(continued)

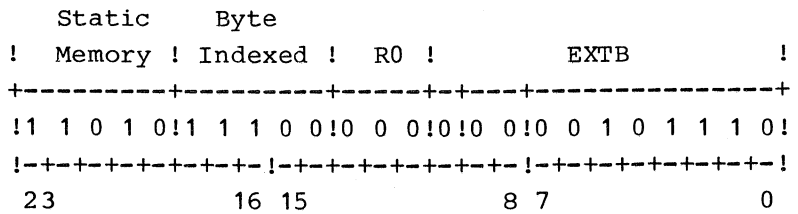
The next extensions to be appended are the addressing extensions required by the addressing modes for the general operands. Since Operand A is specified using the Static Memory addressing mode, it requires one displacement field, containing 10. This displacement is placed in single-byte format after the Index Byte.

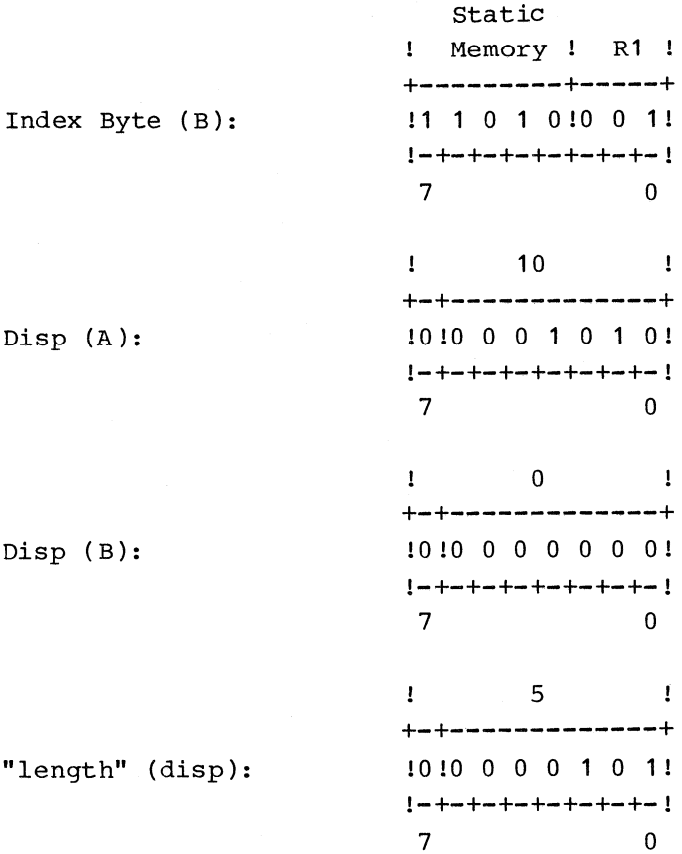
The Static Memory basemode 0(SB) for Operand B requires one displacement field containing 0. This displacement is placed in single-byte format after the displacement field for Operand A.



(continued)

Finally, the length operand (specified as 5) is an implied displacement which is appended after all addressing extensions. It also can be encoded in single-byte format due to its small contents. Thus, the complete machine instruction is:





The complete binary form of this instruction therefore appears in consecutive memory bytes as

2E 00 D7 D1 0A 00 05 (Hex).

Chapter 5

SERIES 32000 INSTRUCTION SET

This chapter contains the detailed definitions of each of the instructions in the Series 32000 instruction set.

Instructions are presented in the format shown in Figure 5-1. The items indicated there are described below.

1. Mnemonic index. Instructions are alphabetized according to this index, which gives a general form of the mnemonic(s) for each instruction. For a listing of instructions by functional groups, see instead Appendix A or Chapter 3.

2. Enumerated mnemonics. This area holds a list of all valid mnemonic forms for the instruction, if there are alternative forms.

3. Format definition. This area defines the assembly-language and binary formats of the instruction, and the number and kinds of operands. The information contained here is explained in Chapter 4.

4. Instruction description. The operation performed by the instruction is defined here.

5. Flags Affected. All flags in the Processor Status Register which are affected by the instruction are listed. See Section 2.2 for the general definitions of these flags.

6. Traps. Any trap that may be caused by the instruction is listed. See Chapter 6 for details of interrupt and trap service.

   NOTE: Since the Abort trap, Trap (ABT), may occur on any instruction for memory management purposes, it is not listed unless there is a cause which is unique to that instruction.

7. Examples. One or more examples are given, where required, in order to clarify the operation performed by the instruction. Conventions used in presenting example instructions and operands are given in Section 5.1.

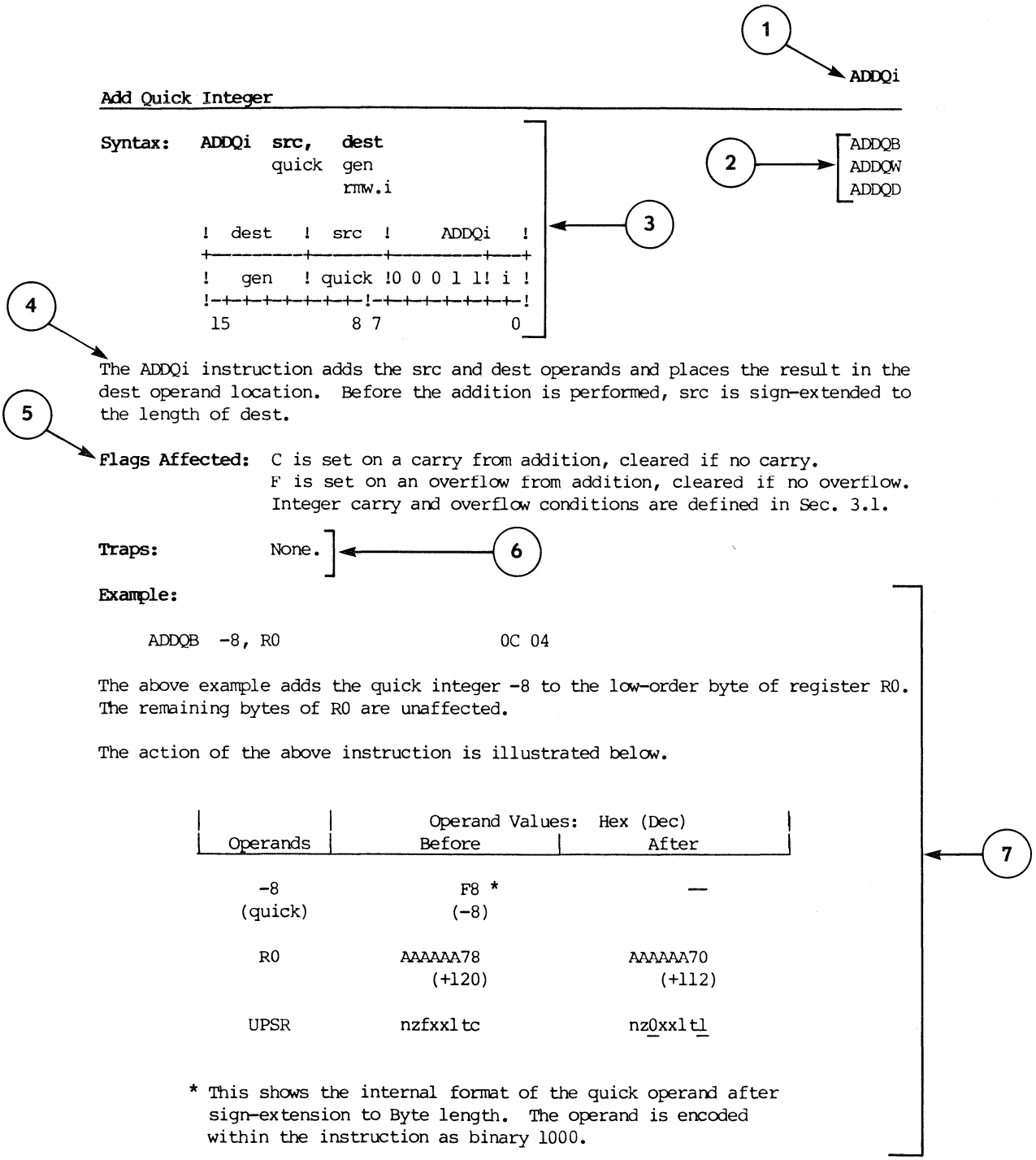


Figure 5-1 Typical Instruction Definition

5.1 Instruction Examples

Figure 5-2 shows an instruction example from Section 5.2. Each example shows the encodings and the actions of one or more typical forms of the instruction being described.

5.1.1 Coding Examples

Example instructions are shown coded both in assembly-language source form and in machine-language form.

The machine-language form is presented in hexadecimal as would be expected in a "dump" format. The leftmost byte displayed occupies the lowest memory address. The entire instruction is presented, including all extensions.

5.1.2 Action Examples

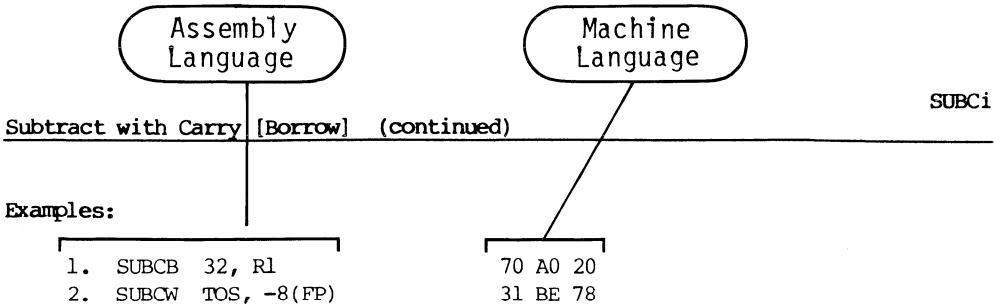
The actions of an example instruction are shown in three columns.

The "Operands" column identifies all operands of the instruction: both those explicitly stated in assembly language and those which are implicitly affected by "side-effects" (e.g. the PSR and SP registers where relevant). When a number is presented it generally refers to an operand at that memory address, and is a hexadecimal value. However, if the comment "(immediate)" or "(disp)" appears below it, it is a literal value provided from within the instruction itself, and is presented symbolically as in the assembly-language form of the instruction. Its value appears in the "Before" column.

The "Before" and "After" columns present the values of operands before and after execution of the example instruction. The radixes used in presenting these values are listed in the column heading, as

"Hex" = Hexadecimal,
"Binary" = Binary,
"Boolean" = Boolean interpretation of the value (True or False), or
"Dec" = Decimal interpretation of the value. Where a value can be interpreted as either signed or unsigned, and the distinction is relevant to the action of the instruction, the terms "Signed" and "Unsigned" are used.

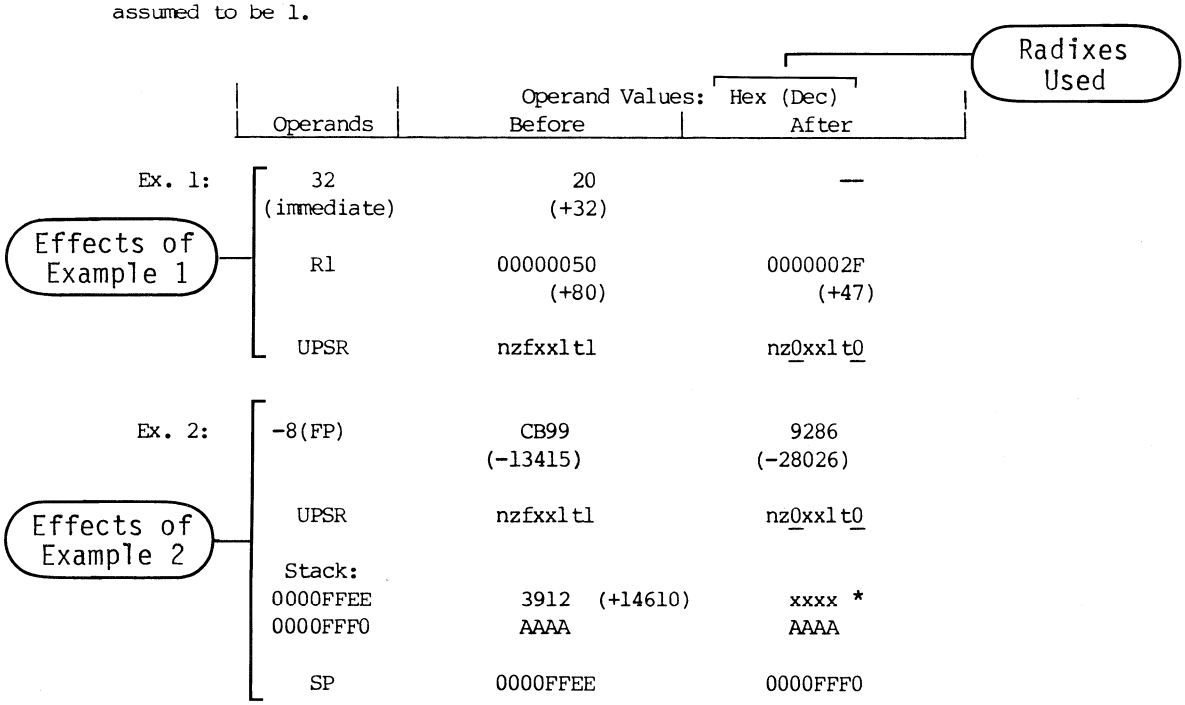
NOTE: An immediate or displacement value is not considered to have an "After" value, even though it never changes, because it is not available as an immediate or displacement value to any subsequent instructions.



Example 1 subtracts the sum of 32 and the C flag value from the low-order byte of register R1 and places the result in the low-order byte of register R1. The remaining bytes of R1 are not affected.

Example 2 subtracts the sum of the word at the top of the stack and the C flag value from the word at the memory address specified by -8(FP). The instruction then places the two-byte result at the memory address specified as -8(FP).

The actions of the above instructions are illustrated below. The C flag value is assumed to be 1.



* The instruction has not itself changed the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

Figure 5-2 Typical Instruction Example

5.1.3 Operand Presentation Format

The memory format convention used by the Series 32000 family places the least-significant byte of a memory operand at the first (i.e. lowest) address. The correct interpretation of a multiple-byte value in memory, therefore, is produced by assembling consecutive bytes of the value from right to left. The address of an operand in memory is also the address of its least-significant byte.

Operand values in examples are presented in units of bytes, words, double-words or quad-words. Each unit is shown in the form corresponding to the interpretation of its contents, so that the least-significant digit of its least-significant byte always appears as the rightmost digit.

Units appearing consecutively in memory are separated from each other either horizontally (by a space) or vertically. Memory addresses of consecutive units increase to the right and downward. The value given in the Operand column is the address of the first unit (i.e. the address of its least-significant byte). For example,

```
5000      1234 5678 9ABC      and      5000      1234
                                                 5678
                                                 9ABC
```

both show three consecutive 16-bit words in memory starting with the value 1234 at address 5000. If the same memory information were presented as consecutive bytes, it would appear as

```
5000      34 12 78 56 BC 9A .
```

Because an immediate or displacement value is encoded within the instruction format with its most-significant byte at the lowest address (i.e. backward from the ordering used elsewhere in memory), any such value is presented in the form of consecutive bytes.

Hexadecimal and binary operand representations are always presented fully, including any leading zeroes, in order to define the length of each unit unambiguously.

The character "x" means "don't care". Within a value in the Before column, any field made up of these characters is ignored. Within a result in the After column, these represent a field which may be changed unpredictably. In a binary value, each "x" represents one don't care bit. In a hexadecimal value, each "x" represents four bits, all of which are don't care bits.

Filler values of hexadecimal A...A, B...B or C...C are used in examples instead of x...x whenever there is information which is ignored but also not changed. Any decimal interpretation given with the operand ignores these fields. The values 0...0 and F...F are never used as filler, as they occur very often within the significant portion of an operand.

The Processor Status register (PSR) is presented in binary, in the form xxxxIPSU/NZFxxLTC. In the Before column of an example, lower-case letters (e.g. xxxxipsu/nzfxxltc) represent initially unknown values of the corresponding bits. Any bits appearing in the After column which still contain these lower-case symbols have not been changed by the instruction being illustrated, with the exception of all bits shown as "x", which are don't care bits as defined above. Any bits which are changed by the instruction are shown in the After column with their new values underlined. In situations where the most-significant half of the PSR is never used or affected by an instruction, only the least-significant half of the PSR is shown, labeled UPSR for "User PSR".

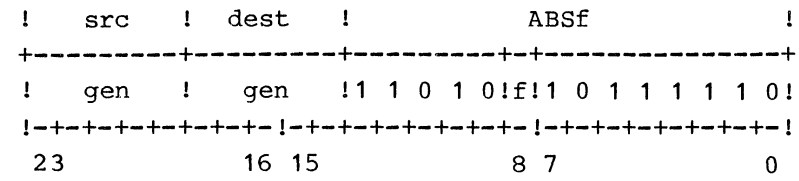
5.2 Instruction Definitions

This section defines the individual Series 32000 instructions. The instructions are ordered alphabetically by their general mnemonic form. For listings of instructions by functional groups, see Appendix A. For help in interpreting the information presented here, see the beginning of this chapter.

ABSf

Absolute Value Floating

Syntax: **ABSf** **src,** **dest** ABSF
gen gen ABSL
read.f write.f



The ABSf instruction computes the absolute value of the src operand and places the result in the dest operand location.

Flags Affected: No PSR flags.
The FSR TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes. See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap(UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Example:

```
ABSF  F0, F2                    BE B5 00
```

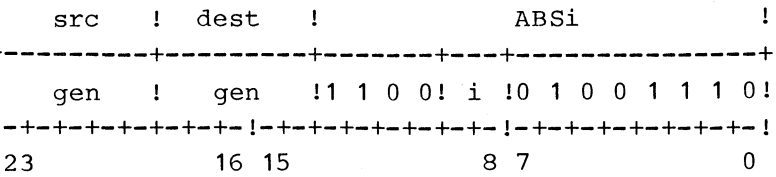
This example computes the absolute value of the single-precision number in register F0 and places the result in register F2.

The instruction is illustrated below:

Operands	Operand Values: Hex (Dec) Before	After
F0	C2250000 (-41.25)	C2250000 (-41.25)
F2	AAAAAAAA	42250000 (+41.25)

Absolute Value

Syntax: **ABSi** **src,** **dest** ABSB
gen gen ABSW
read.i write.i ABSD



The ABSi instruction computes the absolute value of the src operand and places the result in the dest operand location.

The absolute value of a positive number is the number itself. The absolute value of a negative number is taken by subtracting it (as two's complement) from zero.

Flags Affected: F is set if an overflow from subtraction occurs, cleared otherwise. An overflow condition will occur if the src operand is the most negative number that can be represented in the operand length specified by the programmer. For bytes, this value is -128 (Hex 80); for words it is -32768 (Hex 8000) and for double-words it is -2,147,483,648 (Hex 80000000). These values have no corresponding positive values in the same operand length. The result produced on an overflow is the original src operand value.

C is not affected.

Traps: None.

Examples:

```
1.  ABSB  R5, R6                4E B0 29
2.  ABSD  8(SP), R7             4E F3 C9 08
```

Example 1 computes the absolute value of the low-order byte of register R5 and places the result in the low-order byte of register R6. The remaining bytes of R6 are not affected.

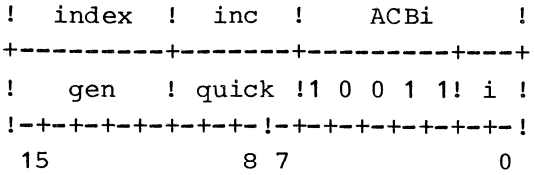
Example 2 computes the absolute value of the double-word at the memory address specified by 8(SP) and places the result in register R7.

These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R5	AAAAAA13 (+19)	AAAAAA13 (+19)
	R6	BBBBBBBB	BBBBBB13 (+19)
	UPSR	nzfxxltc	nz0xxltc
Ex. 2:	8(SP)	FFFFFFFF (-1)	FFFFFFFF (-1)
	R7	AAAAAAAA	00000001 (+1)
	UPSR	nzfxxltc	nz0xxltc

Add, Compare and Branch

Syntax: **ACBi** **inc, index, dest** ACBB
quick gen disp ACBW
rmw.i ACBD



The ACBi instruction adds the inc value to the index operand (after sign- extending the 4-bit inc value to the length of index) and places the sum in the index operand location. If the sum is not zero, the instruction branches to the location specified as dest. If the sum is zero, the instruction ignores dest and passes control to the next sequential instruction.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction. Using the NSC Series 32000 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or as any addressing expression that evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Example:

```
LOOP:   MULD  R2, R1              CE 63 10
        ACBB -1, R0, LOOP         CC 07 7D
```

In this example, the ACBB instruction adds -1 to the low-order byte of register R0 and passes execution control to the MULD statement labeled LOOP as long as the result is not zero. The combined instructions form an iterative loop.

The action of each execution of the above ACBB instruction is illustrated below. Initial values for registers R0, R1, and R2 are assumed to be 3, 2, and 2, respectively. Note that at the first execution of the ACBB instruction the first MULD instruction has already been executed. The MULD instruction, labeled LOOP, is assumed to be at address 9000 Hex, and the ACBB instruction is assumed to be at address 9003 Hex.

		Operand Values: Hex	
	Operand	Before	After
1:	PC	00009003	00009000 *
	R0	AAAAAA03	AAAAAA02
	R1	00000004	00000004
	R2	00000002	00000002
2:	PC	00009003	00009000 *
	R0	AAAAAA02	AAAAAA01
	R1	00000008	00000008
	R2	00000002	00000002
3:	PC	00009003	00009006 **
	R0	AAAAAA01	AAAAAA00
	R1	00000010	00000010 ***
	R2	00000002	00000002

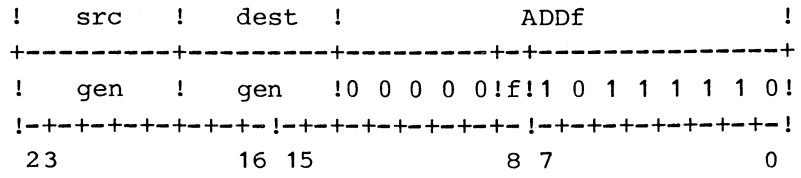
* The disp operand value is assumed to be -3, encoded in one-byte displacement format as 7D Hex. This is the difference between the statement labeled LOOP and the ACBB instruction.

** The ACBB instruction is executed three times and returns control to the MULD instruction at address 9000 twice. At the third execution, register R0 is decremented to zero so the instruction passes control to the next sequential instruction at address 9006.

*** The final result of the MULD iterative loop is ((2*2)*2)*2 or 16 (=10 Hex).

Add Floating

Syntax: **ADDf** **src,** **dest** ADDF
gen gen ADDL
read.f rmw.f



The ADDf instruction adds the src and dest operands and places the result in the dest operand location. Results for normalized and zero operands are given in the table below. The symbols "m" and "n" represent any non-zero normalized numbers. The symbols "+z" and "-z" represent positive zero and negative zero, respectively.

dest: / src	n	+z	-z
m	m+n*	m	m
+z	n	+z	*
-z	n	*	-z

* These cases, when the result is zero, select the result based on the current rounding mode selected in the FSR. If the "Round toward Negative Infinity" mode is selected, then the result returned is negative zero. Otherwise, the result returned is positive zero.

Flags Affected: No PSR flags. FSR flags are affected as follows:
UF is set if an underflow occurs; unaffected otherwise.
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Examples:

```
1.  ADDF  F0, F7                BE C1 01
2.  ADDL  F2, 16(SB)            BE 80 16 10
```

Example 1 adds the single-precision numbers in registers F0 and F7 and places the result in register F7.

Example 2 adds the double-precision numbers in register pair (F2,F3) and at the address 16(SB) and places the double-precision result at address 16(SB).

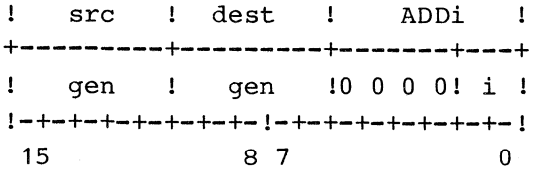
	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	40840000 (+4.125)	40840000 (+4.125)
	F7	41D40000 (+26.5)	41F50000 (+30.625)
Ex. 2:	(F2,F3)	41C0200888300000 (+541069584.375)	41C0200888300000 (+541069584.375)
	16(SB)	4114C86300000000 (+340504.75)	41C022A194900000 (+541410089.125)

Add

Syntax: **ADDi** **src,** **dest**

	gen	gen
	read.i	rmw.i

ADDB
ADDW
ADDD



The ADDi instruction adds the src and dest operands and places the sum in the dest operand location.

Flags Affected: C is set on a carry from addition, cleared if no carry.
F is set on an overflow from addition, cleared if no overflow.
Integer carry and overflow conditions are defined in Section 3.1

Traps: None.

Examples:

```
1.  ADDB  R0, R1                    40 00
2.  ADDD  4(SB), -4(FP)             03 D6 04 7C
```

Example 1 adds the low-order byte of register R0 to the low-order byte of register R1 and places the result in the low-order byte of register R1. The remaining bytes of R1 are not affected.

Example 2 adds double-words. 4(SB) and -4(FP) specify the operand addresses. The instruction places the double-word sum in memory at the address specified by -4(FP).

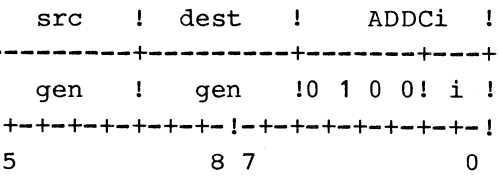
	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R0	AAAAAA9F (-97)	AAAAAA9F (-97)
	R1	BBBBBB62 (+98)	BBBBBB01 (+1)
	UPSR	nzfxxltc	nz0xxlt1
Ex. 2:	4(SB)	20401110 (+541069584)	20401110 (+541069584)
	-4(FP)	0334A001 (+53780481)	2374B111 (+594850065)
	UPSR	nzfxxltc	nz0xxlt0

Add with Carry

Syntax: **ADDCi** **src,** **dest** ADDCB

gen gen ADDCW

read.i rmw.i ADDCD



The ADDCi instruction adds the src operand, dest operand, and the C flag and places the sum in the dest operand location.

Flags Affected: C is set on a carry from addition, cleared if no carry.
F is set on an overflow from addition, cleared if no overflow.
Integer carry and overflow conditions are defined in Section 3.1.

Traps: None.

Examples:

```
1.  ADDCB   32, R0              10 A0 20
2.  ADDCD   8(SB), R0           13 D0 08
```

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

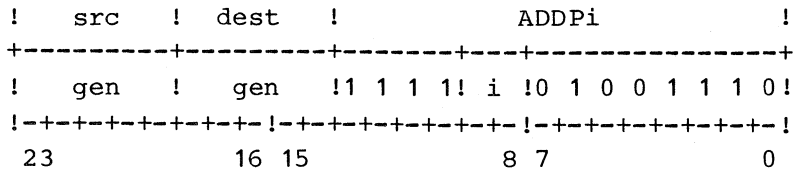
Example 2 adds the double-word at the address specified by 8(SB), the contents of the register R0, and the contents of the C flag and places the result in register R0.

In the following illustration, the C flag is assumed to be 1.

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	32 (immediate)	20 (+32)	--
	R0	AAAAAA0F (+15)	AAAAAA30 (+48)
	UPSR	nzfxxlt1	nz0xxlt0
Ex. 2:	8(SB)	FFFFFFFF (-1)	FFFFFFFF (-1)
	R0	00000030 (+48)	00000030 (+48)
	UPSR	nzfxxlt1	nz0xxlt1

Add Packed Decimal

Syntax: **ADDPi** **src,** **dest** ADDPB
gen gen ADDPW
read.i rmw.i ADDPD



The ADDPi instruction adds the src operand, dest operand, and the C flag and places the result in the dest operand location as a packed decimal (BCD) integer.

The src and dest operands are interpreted as unsigned packed decimal (BCD) integers. If either operand contains invalid digits, the result is undefined. See Section 3.2 for details of packed decimal arithmetic.

Flags Affected: C is set on a carry, cleared if no carry.
F is cleared.
The packed decimal carry condition is defined in Section 3.2.

Traps: None.

Examples:

```
1.  ADDPD   R0, R1                  4E 7F 00
2.  ADDPB   5(SB), TOS              4E FC D5 05
```

Example 1 adds the packed decimal double-word integers contained in registers R0 and R1 and the C flag and places the result in register R1.

Example 2 adds two byte-long packed decimal integers. The integers are at the addresses specified by 5(SB) and TOS. The instruction places the one-byte result on the top of the stack.

In the following illustrations, the C flag value is assumed to be 0.

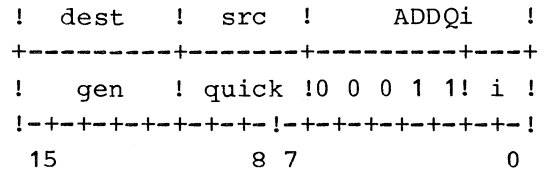
	Operands	Operand Values: Hex * Before	After
Ex. 1:	R0	75308643	75308643
	R1	12345678	87654321
	UPSR	nzfxxlt0	nz0xxlt0
Ex. 2:	5(SB)	99	99
	SP	0000FFDE	0000FFDE
	Stack:		
	0000FFDE	01	00 **
	0000FFDF	AA	AA
	UPSR	nzfxxlt0	nz0xxlt1

* The hexadecimal representation also expresses the decimal interpretation of the value.

** In Example 2, a carry occurs.

Add Quick Integer

Syntax: **ADDQi** **src,** **dest** ADDQB
quick gen ADDQW
rmw.i ADDQD



The ADDQi instruction adds the src and dest operands and places the result in the dest operand location. Before the addition is performed, src is sign-extended to the length of dest.

Flags Affected: C is set on a carry from addition, cleared if no carry.
F is set on an overflow from addition, cleared if no overflow.
Integer carry and overflow conditions are defined in Section 3.1.

Traps: None.

Example:

```
ADDQB  -8, R0                    0C 04
```

The above example adds the quick integer -8 to the low-order byte of register R0. The remaining bytes of R0 are unaffected.

The action of the above instruction is illustrated below.

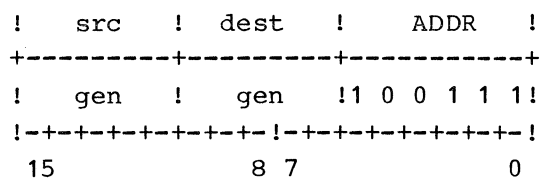
Operands	Operand Values: Hex (Dec) Before	After
-8 (quick)	F8 * (-8)	--
R0	AAAAAA78 (+120)	AAAAAA70 (+112)
UPSR	nzfxxltc	nz0xxlt1

* This shows the internal format of the quick operand after sign-extension to byte length. The operand is encoded within the instruction as binary 1000.

ADDR

Compute Effective Address

Syntax: **ADDR** **src,** **dest**
gen gen
addr write.D



The ADDR instruction places the effective address of the src operand into the dest operand location. The src operand itself is not referenced.

NOTE: If the ADDR instruction is used with the External addressing mode for an external procedure, it does not generate the effective address of the procedure entry point, but it can be used instead to copy the procedure's descriptor from the current Link Table into the dest operand location. The NSC Series 32000 assembler provides an alternative mnemonic LXPD (Load External Procedure Descriptor, q.v.) for this use.

Flags Affected: None.

Traps: None.

Example:

```
ADDR  4(FP), R0                    27 C0 04
```

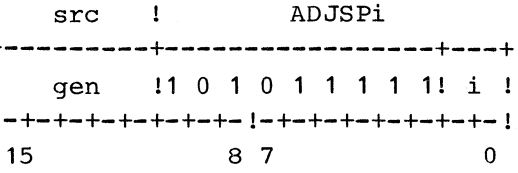
This example places the effective address specified as 4(FP) into register R0.

	Operand Values: Hex	
Operands	Before	After
FP	00001000	00001000
R0	AAAAAAAA	00001004 *

* The effective address of 4(FP) is the sum of the contents of the FP register (H'1000) and the displacement 4, as defined for the Frame Memory addressing mode.

Adjust Stack Pointer

Syntax: **ADJSPi** **src** ADJSPB
gen ADJSPW
read.i ADJSPD



The ADJSPi instruction adjusts the value of the current stack pointer by subtracting the src operand from it. This has the effect of lengthening the stack by the number of bytes given in the src operand if positive, and shortening it if src is negative. The S flag in the PSR determines whether the current stack pointer register is SP0 or SP1. Regardless of the length of the src operand, the entire stack pointer is modified. The src operand is interpreted as a signed integer, and is sign-extended to 32 bits before the subtraction is performed.

Flags Affected: None.

Traps: None.

Example:

```
ADJSPD  -4(FP)                    7F C5 7C
```

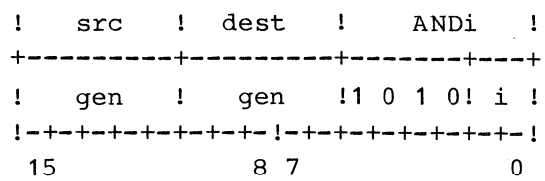
This instruction subtracts the double-word at address -4(FP) from the contents of the current stack pointer, lengthening the stack by that number of bytes.

In the following illustration, the PSR S flag is assumed to be set, selecting register SP1 as the current stack pointer.

	Operand Values: Hex	
Operands	Before	After
-4(FP)	00000010	00000010
SP1	00001010	00001000

ANDi
And

Syntax: **ANDi** **src,** **dest** ANDB
gen gen ANDW
read.i rmw.i ANDD



The ANDi instruction performs a bit-wise logical AND on the src and dest operands and places the result in the dest operand location.

The instruction ANDs each bit in src with the corresponding dest bit. If two corresponding bits are both "1", the dest bit is set to "1"; otherwise, the dest bit is set to "0".

Flags Affected: None.

Traps: None.

Example:

```
ANDB  R0, R1                    68 00
```

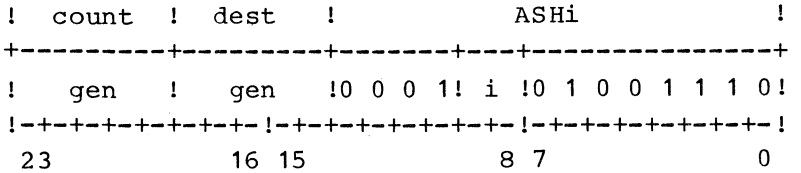
This example ANDs the low-order bytes of registers R0 and R1 and places the result in the low-order byte of R1. The instruction affects only the low-order byte of R1.

The instruction is illustrated below:

Operands	Operand Values: Binary Before	After
R0 (low byte)	10010010	10010010
R1 (low byte)	01110111	00010010

Arithmetic Shift

Syntax: **ASHi** **count,** **dest** ASHB
gen gen ASHW
read.B rmw.i ASHD



The ASHi instruction performs an arithmetic shift on the dest operand in the manner specified by the count operand. The sign of count determines the direction of the shift. The absolute value of count gives the number of bit positions to shift the dest operand.

The count operand value must be within the range -7 to +7 for the ASHB form, -15 to +15 for the ASHW form, and -31 to +31 for the ASHD form. A positive count specifies a left shift; a negative count specifies a right shift. In an arithmetic left shift, high-order bits (including the sign bit) shifted out of dest are lost, and low-order bit positions emptied by the shift are zero-filled. In an arithmetic right shift, low-order bits shifted out of dest are lost, and all high-order bit positions emptied by the shift are filled from the original sign bit of dest.

The count and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: None.

Examples:

```
1.  ASHB  2, 16(SB)          4E 84 A6 02 10
2.  ASHB  TOS, 16(SB)        4E 84 BE 10
```

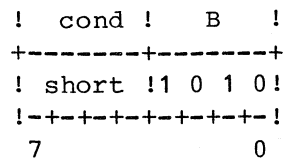
Example 1 shifts the byte specified by 16(SB) two bit positions to the left.

Example 2 pops a byte from the top of the currently-selected stack. Based on this value, it shifts the byte specified by 16(SB) accordingly.

	Operands	Operand Values: Binary (Dec) Before	After
Ex. 1:	2 (immediate)	00000010 (+2)	--
	16(SB)	00011111 (+31)	01111100 (+124)
Ex. 2:	Stack:		
	(48000)	11111110 (-2)	xxxxxxxx
	(48001)	10101010	10101010
	16(SB)	11111000 (-8)	11111110 (-2)
	SP	(48000)	(48001)

Conditional Branch

Syntax: **Bcond** **dest**
disp



The Bcond instruction branches to the location specified as dest if the condition specified by cond is true. If the condition is false, execution continues with the next sequential instruction.

Cond is a two character condition name that specifies the state of a flag or flags in the PSR. If the flag(s) have the specified state, the condition is true; otherwise, the condition is false.

The Conditional Branch instruction may specify the following conditions:

Condition	Condition Name	True State	Short Field
Equal	EQ	Z flag set	0000
Not Equal	NE	Z flag clear	0001
Carry Set	CS	C flag set	0010
Carry Clear	CC	C flag clear	0011
Higher	HI	L flag set	0100
Lower or Same	LS	L flag clear	0101
Greater Than	GT	N flag set	0110
Less Than or Equal	LE	N flag clear	0111
Flag Set	FS	F flag set	1000
Flag Clear	FC	F flag clear	1001
Lower	LO	Z and L flags clear	1010
Higher or Same	HS	Z or L flag set	1011
Less Than	LT	Z and N flags clear	1100
Greater Than or Equal	GE	Z or N flag set	1101

The condition name is appended to the instruction mnemonic as illustrated in the following examples. The name is translated at assembly time to the corresponding 4-bit Short field of the basic instruction.

The interpretation of condition codes is such that the instruction sequence

```
CMPB    A,B
BGT     ERROR
```

will cause a branch if operand A is greater than operand B in the CMPB instruction.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction (see Section 4.2.3 for displacement formats). Using the ASM16 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or any addressing expression which evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Examples:

1.	BLO LOOP	AA BF 66
2.	BNE *+10	1A 0A

Example 1 passes execution control to the instruction labeled LOOP if the Z and L flags in the PSR are 0.

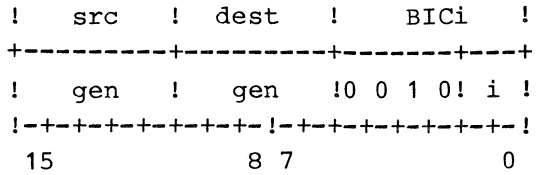
Example 2 passes execution control to a nonsequential instruction if the Z flag is 0. The instruction passes execution control by adding 10 to the PC register.

In the following illustrations, the Z and L flags are assumed to be zero. LOOP is assumed to be the label of a statement beginning at address 9000 Hex.

	Operand	Operand Values: Hex (Dec) Before	After
Ex. 1:	PC	0000909A (37018)	00009000 (36864)
	LOOP (disp)	BF 66 (-154)	--
	UPSR	n0fxx0tc	n0fxx0tc
Ex. 2:	PC	00009FF0 (40944)	00009FFA (40954)
	*+10 (disp)	0A (+10)	--
	UPSR	n0fxx0tc	n0fxx0tc

Bit Clear

Syntax: **BICi** **src,** **dest** BICB
gen gen BICW
read.i rmw.i BICD



The BICi instruction clears (sets to 0) the bits in the dest operand that correspond to the "1" bits in the src operand.

Flags Affected: None.

Traps: None.

Example:

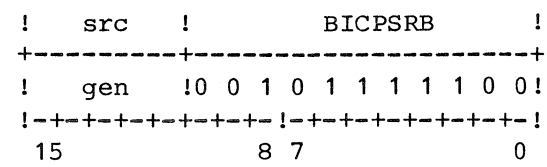
```
BICB  R0, 3(SB)                    88 06 03
```

This example clears the bits, in the byte at address 3(SB), corresponding to the "1" bits in the low-order byte of register R0.

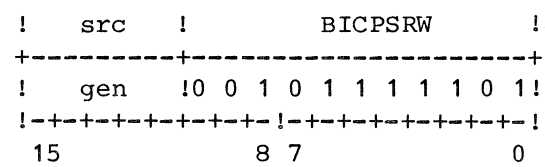
Operands	Operand Values (Binary) Before	After
R0 (low byte)	10011001	10011001
3(SB)	11110000	01100000

BICPSRB
BICPSRW
Bit Clear in PSR

Syntax: **BICPSRB** **src**
gen
read.B



Syntax: **BICPSRW** **src**
gen
read.W



The Bit Clear in PSR instructions clear (set to 0) the bits in the PSR corresponding to the "1" bits in the src operand. The BICPSRB instruction affects only the low-order byte of the PSR; the BICPSRW instruction affects the entire PSR.

Flags Affected: Flags specified by src "1" bits are cleared.

Traps: Illegal Operation Trap (ILL) is activated if a BICPSRW instruction is attempted while the PSR U flag is set.

Example:

```
BICPSRB  B'10100010                7C A1 A2
```

This instruction clears bits 1, 5 and 7 in the low-order byte of the PSR. These are the T, F and N flags, respectively.

The instruction is illustrated below:

Operands	Operand Values (Binary) Before	After
B'10100010 (immediate)	10100010	--
PSR	xxxxipsu/nzfxxltc	xxxxipsu/0z0xxl0c

Bit Set in PSR

Syntax: **BISPSRB** **src**
gen
read.B

```
!   src   !        BISPSRB        !
+---------+-----------------------+
!   gen   !0 1 1 0 1 1 1 1 1 1 0 0!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 15              8 7              0
```

Syntax: **BISPSRW** **src**
gen
read.W

```
!   src   !        BISPSRW        !
+---------+-----------------------+
!   gen   !0 0 1 0 1 1 1 1 1 1 0 1!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 15              8 7              0
```

The BISPSRB and BISPSRW instructions set the bits in the PSR corresponding to the "1" bits in the src operand.

Flags Affected: Flags specified by src "1" bits are set.

Traps: Illegal Operation Trap (ILL) is activated if a BISPSRW instruction is attempted while the PSR U flag is 1.

Example:

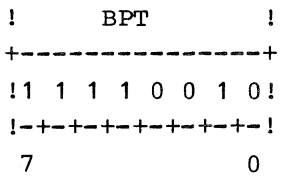
```
BISPSRB  B'10100010              7C A3 A2
```

This instruction sets bits 1, 5 and 7 in the low-order byte of the PSR. These are the T, F and N flags, respectively.

	Operand Values (Binary)	
Operands	Before	After
B'10100010 (immediate)	10100010	--
PSR	xxxxipsu/nzfxxltc	xxxxipsu/1z1xxl1c

BPT
Breakpoint Trap

Syntax: **BPT**



The BPT instruction activates the Breakpoint Trap (BPT). See "Exceptions", Chapter 6. The return address pushed on the Interrupt Stack is the address of the BPT instruction itself.

Flags Affected: None.

Traps: Breakpoint Trap (BPT) is activated.

Example:

```
BPT                                F2
```

Unconditional Branch

Syntax: **BR** **dest**
disp

```
!       BR        !
+-----------------+
!1 1 1 0 1 0 1 0!
!-+-+-+-+-+-+-+-!
 7               0
```

The BR instruction branches to the location specified as dest.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction. Using the NSC Series 32000 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or any addressing expression which evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Examples:

1.	BR ERROR	EA BF 66
2.	BR *+10	EA 0A

Example 1 passes execution control to the instruction labeled ERROR.

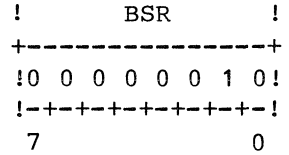
Example 2 passes execution control to a nonsequential instruction by adding 10 to the PC register.

In the following illustration, ERROR is assumed to be the label of a statement beginning at address 9000 Hex.

	Operand	Operand Values: Hex (Dec) Before	After
Ex. 1:	PC	0000909A (37018)	00009000 (36864)
	ERROR (disp)	BF 66 (-154)	--
Ex. 2:	PC	00009FF0 (40944)	00009FFA (40954)
	*+10 (disp)	0A (+10)	--

BSR
Branch To Subroutine

Syntax: **BSR** **dest**
disp



The BSR instruction calls the local procedure at the address specified as dest. It does so by pushing the address of the next sequential instruction onto the currently-selected stack and branching.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction. Using the NSC Series 32000 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or any addressing expression which evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Example:

```
BSR CALC                                02 10
```

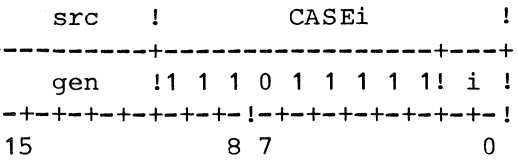
This example causes a program to branch to the local procedure labeled CALC after saving the address of the next sequential instruction on the stack.

The action of the above instruction is illustrated below. The statement labeled CALC is assumed to be at address 9010 Hex, 16 bytes forward from the first byte of this instruction. Since the displacement field is one byte long, the total length of this instruction is two bytes. The return address is therefore the original PC contents plus two.

Operand	Operand Values: Hex (Dec) Before	After
PC	00009000	00009010
CALC (disp)	10 (+16)	--
SP	0000FFD4	0000FFD0
Stack:		
0000FFD0	xxxxxxxx	00009002
0000FFD4	AAAAAAAA	AAAAAAAA

Case Branch

Syntax: **CASEi** **src** CASEB
gen CASEW
read.i CASED



The CASEi instruction branches to a nonsequential instruction by adding the src operand to the PC register. The src operand is interpreted as a signed integer, and is sign-extended to 32 bits before the addition is performed.

A Case Branch instruction, using Scaled Indexing and a table of branch offsets, may be used to implement a multiway branch. See example below.

Flags Affected: None.

Traps: None.

Example:

```
            CASEB  TABLE[R7:B]       7C E7 DF 04
TABLE:      ; (starts here)
```

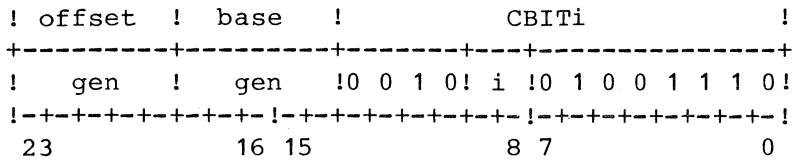
This example branches to a nonsequential instruction by adding the byte at the address specified by TABLE[R7:B] to the PC register. The entire contents of register R7 determine the location of the operand to be added.

Operand	Operand Values: Hex Before	After
PC	00009000	0000906A
R7	00000005	00000005
TABLE 00009004 *	0A 1A 3A 5A 7A 6A 4A	0A 1A 3A 5A 7A 6A 4A
TABLE[R7:B]	6A	6A

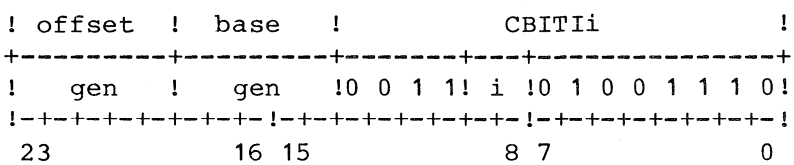
* Address 9004 (Hex) marks the beginning of a table of branch offsets. In this example, the table is located directly after the CASEB instruction, and is accessed via the Program Memory addressing mode, with a displacement of 4. Since register R7 contains 5, the effective address of TABLE[R7:B] is 9009 (Hex). This means the sixth branch offset (6A) is selected to be added to the PC register.

CBITi
CBITIi
Clear Bit, Clear Bit Interlocked

Syntax: **CBITi** **offset,** **base** CBITB CBITIB
gen gen CBITW CBITIW
read.i regaddr CBITD CBITID



Syntax: **CBITIi** **offset,** **base**
gen gen
read.i regaddr



The CBITi and CBITIi instructions clear (set to 0) the register or memory bit specified by base and offset after copying the bit value to the F flag in the PSR.

The CBITIB, CBITIW, and CBITID instructions, in addition, activate the Interlocked Operation output pin on the CPU, which may be used in multi-processor systems to interlock accesses to semaphore bits. See the applicable CPU data sheet for further details.

The location of the bit is determined from offset and base. Offset is a general operand, whose length is given by the operation length suffix. Base is an addressing expression giving a byte address from which offset specifies a bit position. See Section 3.5 for details of specifying bit positions.

If base is a register, then the bit is within that register, at the bit position given by the offset operand. If base is a memory location, then the bit is at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.5 for definitions of the operators MOD and DIV above, and for further details of bit instructions.

Offset is interpreted as a signed integer.

Clear Bit (continued)

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

Traps: None.

Example:

```
CBITW  R0, 0(R1)                    4E 49 02 00
```

This example clears a bit in memory after copying the bit value to the F flag. For designating the location of the target bit, the low-order word of register R0 supplies the bit offset, and 0(R1) is specified as the base address.

In the following illustration, the target bit is assumed to be 1 prior to instruction execution.

Operands	Operand Values: Hex (Dec) [Binary] Before	After
R0 (offset)	AAAA004C (+76)	AAAA004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 0(R1)	00001000 (+4096)	--
00001009 * (+4105)	10 [00010000]	00 [00000000]
UPSR	nzfxxltc	nz1xxltc

* The address 1009 (Hex) is the effective address of the byte containing the desired bit. This address is computed from the offset and the base address as follows:

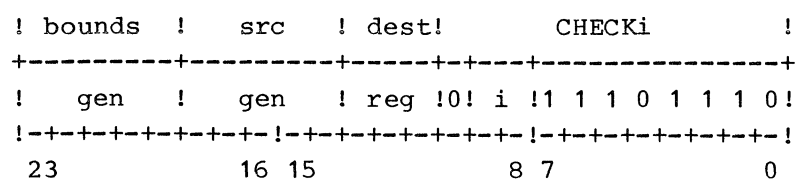
```
base address + (offset DIV 8)
   4096      +       9
   4105 , or 1009 (Hex)  .
```

The bit number within this byte is calculated as:

```
offset MOD 8
  76   MOD 8
     4  .
```

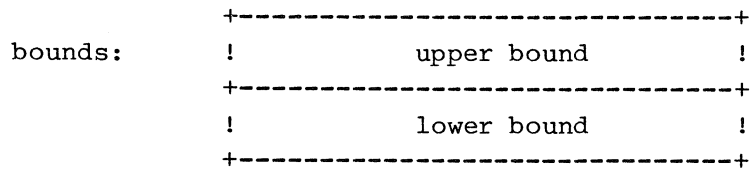
CHECKi
Bounds Check

Syntax: **CHECKi** **dest,** **bounds,** **src** CHECKB
reg gen gen CHECKW
addr read.i CHECKD



The CHECKi instruction compares the src operand against an upper and lower bound from the bounds operand, determining whether it is within those bounds. The instruction then subtracts the lower bound from src, placing the result as a 32-bit value into the general-purpose register specified as dest. This "zero-adjusted" result is usable directly as either an index into a one-dimensional array (within an addressing mode using Scaled Indexing) or as an input value to the INDEX instruction for generating an index into a multi-dimensional array. See Section 3.9 for details of array access.

The bounds operand contains two values--an upper bound followed by a lower bound--as shown:



The upper and lower bounds each have the same length as the src operand. Thus, the entire bounds operand is twice the length of the src operand.

If src is greater than the upper bound or less than the lower bound, it is "out of bounds" and the F flag is set to 1. If src is within the upper and lower bounds, the F flag is cleared.

The instruction places the zero-adjusted result into the dest register. The zero-adjusted value is computed as "src - lower bound". The result is zero-extended to 32 bits. If src is out of bounds, the result placed in dest is undefined.

The src and bounds operands are interpreted as signed integers. The result placed in the dest register is a 32-bit unsigned integer.

Flags Affected: F is set if src is out of bounds, cleared otherwise.

Traps: None.

Bounds Check (continued)

Example:

```
CHECKB  R0, 4(SB), R2              EE 80 D0 04
```

This example compares the low-order byte of register R2 with each of the two one-byte bounds at the address specified by 4(SB). The instruction sets or clears the F flag to indicate the comparison result and then subtracts the lower bound from the low-order byte of R2, placing the result in register R0.

The instruction is illustrated below. An array index in the low-order byte of R2 is being checked against its range of [1..10] and then "zero-adjusted" to its corresponding value for the range [0..9]. The result is being placed into R0 for use in an addressing mode using Scaled Indexing.

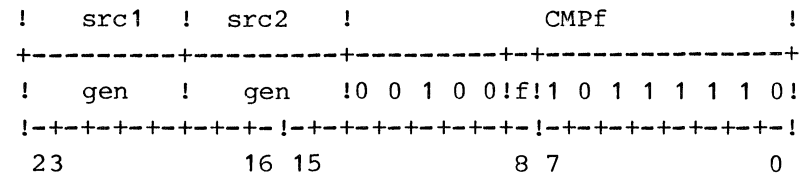
Operands	Operand Values: Hex (Dec) Before	After
R0	AAAAAAAA	00000002 * (+2)
4(SB)	0A 01 (+10,+1)	0A 01 (+10,+1)
R2	AAAAAA03 (+3)	AAAAAA03 (+3)
UPSR	nzfxxltc	nz0xxltc

* The result in R0 represents the result of adjusting the value 3 from a range of [1..10] to the zero-based range of [0..9]. The corresponding adjusted value is 2.

CMPf

Compare Floating

Syntax: **CMPf** **src1,** **src2** CMPF
gen gen CMPL
read.f read.f



The CMPf instruction compares the src1 and src2 operands and sets the Z and N flags to indicate comparison results. Positive and negative zero are equal.

Flags Affected: Z is set if src1 equals src2, cleared otherwise.
N is set if src1 is greater than src2, cleared otherwise.
L is cleared always.
The FSR TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Example:

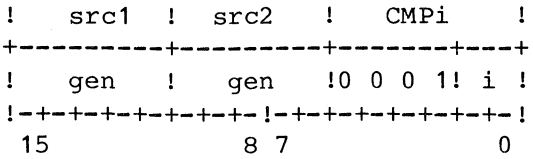
```
CMPF  F0, F2                    BE 89 00
```

This example compares the single-precision numbers in registers F0 and F2.

Operands	Operand Values: Hex (Dec) Before	After
F0	42250000 (+41.25)	42250000 (+41.25)
F2	40A00000 (+5.0)	40A00000 (+5.0)
UPSR	nzfxxltc	10fxx0tc

Compare

Syntax: **CMPi** **src1,** **src2** CMPB
gen gen CMPW
read.i read.i CMPD



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

Flags Affected: Z is set if src1 is equal to src2, cleared otherwise.
N is set if src1 is greater than src2 (signed comparison), cleared otherwise.
L is set if src1 is greater than src2 (unsigned comparison), cleared otherwise.

Traps: None.

Example:

```
CMPB  7(SB), 4(R0)                    04 D2 07 04
```

This example compares byte operands. 7(SB) and 4(R0) specify the operand addresses.

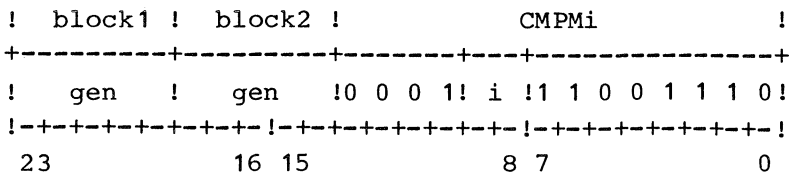
In the following illustration, operand values before instruction execution are assumed; Z, N and L flag values are unknown.

Operands	Operand Values: Hex (Dec) Before	After
7(SB)	FF (signed: -1) (unsigned: +255)	FF (signed: -1) (unsigned: +255)
4(R0)	7F (signed: +127) (unsigned: +127)	7F (signed: +127) (unsigned: +127)
UPSR	nzfxxltc	00fxx1tc

CMPMi

Compare Multiple

Syntax: **CMPMi** **block1,** **block2,** **length** CMPMB
gen gen disp CMPMW
addr addr CMPMD



The CMPMi instruction compares the contents of block1 and block2 and sets the Z, N, and L flags to indicate the comparison result. The blocks are comprised of integers of length i. The number of integers is specified by length.

The instruction compares two integers (one from each block) at a time. If the current integers are equal, the instruction continues with the next two integers; otherwise, the instruction sets the PSR flags and terminates.

The N flag indicates the result of signed integer comparison. The L flag indicates the result of unsigned integer comparison. Both types of comparison are performed.

In assembly language, the length operand is specified as the number of integers in each block. In the machine instruction, however, the length operand is encoded according to the formula

(num - 1) * i

where num is the number of integers in each block, and i is the number of bytes per integer.

A block may not be greater than 16 bytes in length.

Flags Affected: Z is set if block1 and block2 are equal for their entire length; cleared otherwise.

N is set if, in the first unequal pair of integers, the block1 integer is greater than the block2 integer (signed comparison); cleared otherwise.

L is set if, in the first unequal pair of integers, the block1 integer is greater than the block2 integer (unsigned comparison); cleared otherwise.

Traps: None.

Compare Multiple (continued)

Example:

```
CMPMW  10(R0), 16(R1), 4          CE 45 42 0A 10 06
```

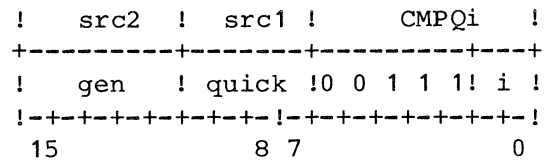
This instruction compares four word-long integers from the block starting at the address specified by 10(R0) to the block starting at the address specified by 16(R1).

Operands	Operand Values: Hex (Signed) [Unsigned] Before	After
R0	00002000	00002000
R1	0000F000	0000F000
0000200A *	1FBE 10A9 8729 (-30935) [+34601] 6511	1FBE 10A9 8729 (-30935) [+34601] 6511
0000F010 **	1FBE 10A9 0839 (+2105) [+2105] 6511	1FBE 10A9 0839 (+2105) [+2105] 6511
UPSR	nzfxxltc	00fxx1tc

* The address of the first block as specified by 10(R0).
** The address of the second block as specified by 16(R1).

CMPQi
Compare Quick Integer

Syntax: **CMPQi** **src1, src2** CMPQB
quick gen CMPQW
read.i CMPQD



The CMPQi instruction compares the src1 and src2 operands and sets the Z, N, and L flags to indicate the comparison result. Before the comparison, src1 is sign-extended to the length of src2. Leading "0"s are supplied for a positive src operand value; leading "1"s for a negative value.

The N flag indicates the result of signed integer comparison. The L flag indicates the result of unsigned integer comparison. Both types of comparison are performed.

Flags Affected: Z is set if src1 is equal to src2, cleared otherwise.
N is set if src1 is greater than src2 (signed comparison), cleared otherwise.
L is set if src1 is greater than src2 (unsigned comparison), cleared otherwise.

Traps: None.

Example:

```
CMPQB  -8, R0                    1C 04
```

This example compares the quick integer -8 with the low-order byte of register R0.

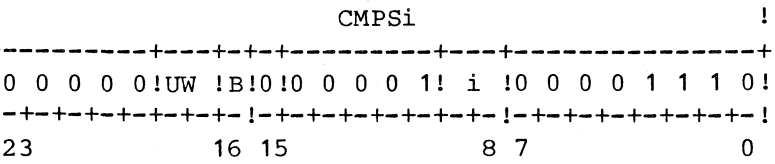
Operands	Operand Values: Hex (Signed) [Unsigned] Before	After
-8 (quick)	F8 * (-8) [+248]	--
R0	AAAAAA00 (0) [0]	AAAAAA00 (0) [0]
UPSR	nzfxxltc	00fxx1tc

* This shows the internal format of the quick operand after sign-extension to Byte length. The operand is encoded within the instruction as binary 1000.

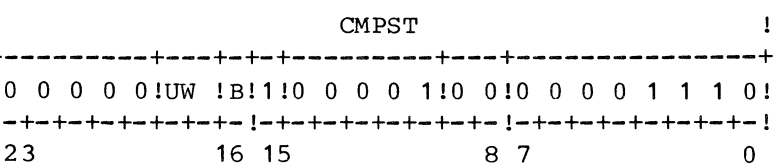
Compare Strings

Syntax: **CMPSi options**

CMPSB
CMPSW
CMPSD
CMPST



Syntax: **CMPST options**



Operands of the CMPSi and CMPST instructions are specified in General Purpose Registers:

R0 - Number of string elements to be processed.
R1 - Address of current String 1 element.
R2 - Address of current String 2 element.
R3 - Address of translation table (CMPST form only).
R4 - Match value (with Until Match or While Match option only).

The CMPSi instruction compares corresponding integer elements from String 1 (address in R1) and String 2 (address in R2) and sets the Z, N and L flags to indicate the comparison results (see "Flags Affected" below). If the current two elements are equal, the instruction compares the next two elements; otherwise, it terminates. After each comparison, the instruction sets register R0 to the number of elements remaining to be compared and sets registers R1 and R2 to the addresses of the next elements to be compared. See Section 3.7 for the exact sequences performed by String instructions.

The N flag indicates the result of signed integer comparison. The L flag indicates the result of unsigned integer comparison. Both types of comparison are performed.

The CMPST instruction compares one-byte elements in String 1, after translation, to one-byte elements in String 2. The translated value to be compared is found by adding the current element from the first string as an unsigned integer to the translation table address found in register R3. The instruction compares elements, sets flags, and sets registers as described above. See Section 3.7 for details of string translation.

Options may be specified by listing the letters B (Backward), U (Until Match) and W (While Match) as operands. The U and W options are mutually exclusive. See Section 3.7 for details of the options available in String instructions.

In the machine instruction, the options are encoded in the B and UW fields as follows:

B field =	0	Forward direction.
	1	Backward direction.
UW field =	00	Neither Until Match nor While Match.
	01	While Match.
	10	(reserved)
	11	Until Match.

String instructions are interruptible. See Section 3.7.

Flags Affected: Z, N and L are affected, as given below.
F is set if the U or W option is specified and the corresponding Until/While condition is met, otherwise it is cleared.

Because of the variety of termination conditions possible in a CMPS instruction, the following sequence is recommended to interpret the flag settings:

1. If the U or W option is specified, then check the F flag. If it is set, then the CMPS instruction has terminated because of the Until Match or While Match test, and the other flag settings are Z=1, N=0, L=0. Register R1 holds the address of the String 1 element which caused termination, and register R2 holds the address of the corresponding element in String 2. Register R0 contains the number of elements left to be processed, including the element which terminated the instruction.

2. If F=0, check the Z flag. If it is set, then the CMPS instruction has terminated because the limit count in R0 has been decremented to zero, and the strings are equal up to that point. Registers R1 and R2 hold the addresses of the next (unprocessed) string elements, and the remaining flag settings are N=0, L=0.

3. If neither the F or Z bit is set (above), then the CMPS instruction has terminated because the strings are unequal. Registers R1 and R2 hold the addresses of the first two string elements that are unequal, and the N and L flags show their relation. Register R0 holds the number of remaining elements, including the element at which the instruction has stopped. If the N bit is set, the element from String 1

Compare Strings (continued)

(indicated by R1) is greater than the element from String 2 (indicated by R2), where both are interpreted as signed integers. If the L bit is set, the element from String 1 is greater than the element from String 2, where both are interpreted as unsigned integers.

Traps: None.

Example:

```
CMPSB                                   0E 04 00
```

This example compares 1-byte elements from two strings until either an unequal pair is found or the limit count in R0 decrements to zero.

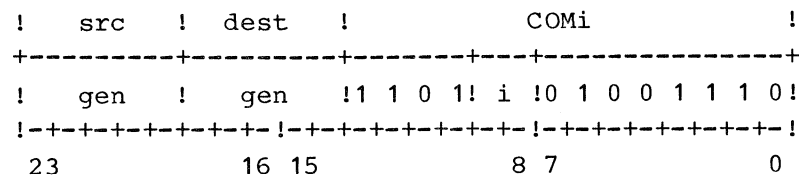
The action of the above instruction on two unequal strings is illustrated below. The underlined string elements show the point at which the instruction terminates.

Operands	Operands Values: Hex (Dec) Before	After
R0	00000020 (+32)	0000000F (+15)
R1	00002000	00002011
R2	0000F000	0000F011
UPSR	nzfxxltc	100xx1tc

Starting Addresses	String Contents
2000	1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
	1F 1E 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18
F000	1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
	1F 1D 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18

COMi
Complement

Syntax: **COMi src, dest** COMB
gen gen COMW
read.i write.i COMD



The COMi instruction places the one's complement of the src operand in the dest operand location.

The one's complement is the logical NOT operation performed on each bit of src. If the src operand bit is "0", then the corresponding dest operand bit is set to "1". Otherwise, the dest bit is set to "0".

Flags Affected: None.

Traps: None.

Example:

```
COMB  R0, -4(FP)                    4E 34 06 7C
```

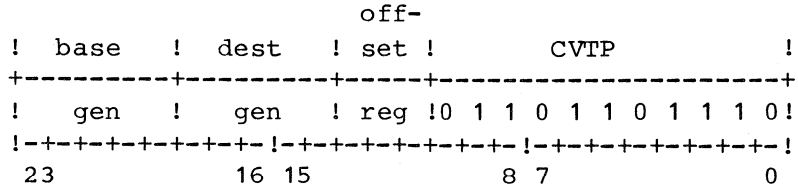
This example places the one's complement of the low-order byte of register R0 in the byte at the address specified by -4(FP).

The instruction is illustrated below:

Operands	Operand Values: Binary Before	After
R0 (low-order)	10101010	10101010
-4(FP)	00000000	01010101

Convert to Bit Pointer

Syntax: **CVTP offset, base, dest**
reg gen gen
addr write.D



The CVTP instruction places in the dest operand location the absolute bit address of the memory bit specified by base and offset. See Section 3.5 for the use of a base and offset in specifying a bit position.

The bit address specifies the number of bits from the first bit in the memory space (bit 0 of the byte at address 0) to the specified bit. The bit address is computed as

8 * EA(base) + offset

where EA(base) is the effective address calculated for base, and offset is a signed byte, word or double-word as given by the operation length.

Flags Affected: None.

Traps: None.

Example:

```
CVTP  R0, 32(SB), R2              6E 83 D0 20
```

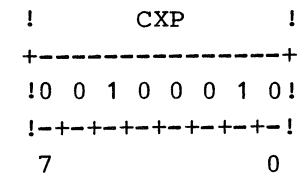
This example computes the absolute bit address of the memory bit specified by register R0 and the address 32(SB). The instruction places the resulting bit address into register R2 as a double-word.

Operands	Operand Values: Hex Before	After
R0	00001234	00001234
SB	00000FE0	00000FE0
base address 32(SB)	00001000	--------
R2	AAAAAAAA	00009234

CXP

Call External Procedure

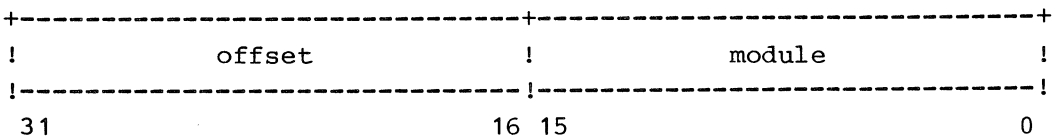
Syntax: **CXP** **index**
disp



The CXP instruction calls a procedure which is outside the current module (an "external" procedure).

The entry point of the external procedure is specified by an external procedure descriptor, which is located in the Link Table (Section 2.7.3) of the current module. The index operand gives the Link Table entry number of the descriptor.

The descriptor is a 32-bit value in the following format:



The descriptor address is the sum of index, multiplied by four, and the contents of the double-word at memory address MOD+4 (the Link Base pointer, Section 2.7.2), where MOD is the contents of the MOD register.

Once the descriptor has been located, the instruction does the following:

1. Decrements the current stack pointer by two, then pushes the contents of the MOD register (16 bits) onto the currently-selected stack. The stack pointer is modified by a total of four in this step. The extra two bytes placed on the stack are reserved for future use.

2. Saves the address of the next sequential instruction (32 bits) onto the currently-selected stack. This double-word is the return address.

3. Copies the low-order word of the descriptor to the MOD register. The low-order word is the address of the new Module Table entry.

4. Copies the double-word at address MOD+0 to the SB register. This double-word is the Static Base pointer for the new module.

Call External Procedure (continued)

5. Copies to the PC register the sum of the high-order word of the descriptor (interpreted as an unsigned value) and the double-word at address MOD+8. This sum is the address of the external procedure entry point in the new module.

Program execution continues at the address placed in the PC register. The procedure has been invoked, and is running in its own module environment.

In the machine instruction, index is encoded as a displacement field appended to the basic instruction. In assembly language, index is specified as the name of the external procedure or in the form of an External addressing mode expression.

Flags Affected: None.

Traps: None.

Examples:

1.	CXP OUTSIDE	22 00
2.	CXP EXT(1)	22 01

Example 1 calls the external procedure named OUTSIDE.

Example 2 calls the external procedure whose descriptor is located in the second entry of the current Link Table (entry number 1).

The action of the instruction in Example 2 is illustrated below:

Operands	Operand Values: Hex Before	After
1 (index)	01	--
90A4 * (descriptor)	00100020 (module 0020) (offset 0010)	00100020
MOD	0010	0020
PC	00009005	0000F010
SB	00009080	0000F100
SP	0000FFF8	0000FFF0
Stack:		
0000FFF0	xxxxxxxx	00009007
0000FFF4	xxxxxxxx	xxxx0010 **
0000FFF8	AAAAAAAA	AAAAAAAA

Module Table:

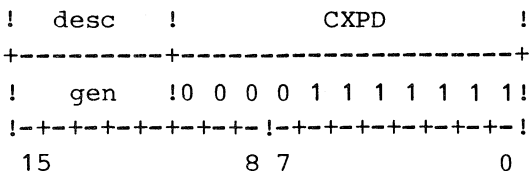
00000010	00009080	(SB)
14	000090A0	(LB)
18	00009000	(PB)
1C	xxxxxxxx	
00000020	0000F100	(SB)
24	0000F110	(LB)
28	0000F000	(PB)
2C	xxxxxxxx	

* 90A4 is the descriptor address. It is computed as the sum of the index 1 in the expression EXT(1), scaled by 4, and the Link Table address. The Link Table address is from address 14 (Hex) in the Module Table.

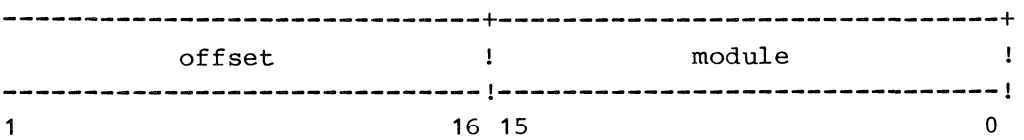
** The 16-bit field shown as "xxxx" is reserved for future use, and should be treated as don't-care bits.

Call External Procedure with Descriptor

Syntax: **CXPD desc**
gen
addr



The CXPD instruction calls the external procedure specified by the desc (descriptor) operand. The descriptor is a 32-bit value in the following format:



The instruction does the following:

1. Decrements the current Stack Pointer by two, then pushes the contents of the MOD register (16 bits) onto the current stack. The stack pointer is modified by a total of four in this step. The extra two bytes placed on the stack are reserved for future use.

2. Saves the address of the next sequential instruction (32 bits) onto the currently-selected stack. This double-word is the return address.

3. Copies the low-order word of the descriptor to the MOD register. The low-order word is the address of the new Module Table entry.

4. Copies the double-word at address MOD+0 to the SB register. This double-word is the Static Base pointer for the new module.

5. Copies to the PC register the sum of the high-order word of the descriptor (interpreted as an unsigned value) and the double-word at address MOD+8. This sum is the address of the external procedure entry point in the new module.

Program execution continues at the address placed in the PC register. The procedure has been invoked, and is running in its own module environment.

Flags Affected: None.

Traps: None.

Example:

```
CXPD  8(SB)                          7F D0 08
```

This example calls an external procedure whose descriptor is contained at memory address 8(SB).

The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
00009088 * (descriptor)	00100020 (module 0020) (offset 0010)	00100020
MOD	0010	0020
PC	00009005	0000F010
SB	00009080	0000F100
SP	0000FFF8	0000FFF0
Stack:		
0000FFF0	xxxxxxxx	00009007
0000FFF4	xxxxxxxx	xxxx0010
0000FFF8	AAAAAAAA	AAAAAAAA

Module Table:

00000010	00009080	(SB)
14	000090A0	(LB)
18	00009000	(PB)
1C	xxxxxxxx	
00000020	0000F100	(SB)
24	0000F110	(LB)
28	0000F000	(PB)
2C	xxxxxxxx	

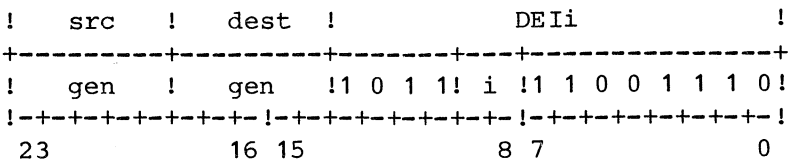
* 9088 (Hex) is the descriptor's effective address, as specified by 8(SB).

** The 16-bit field shown as "xxxx" is reserved for future use, and should be treated as don't-care bits.

Divide Extended Integer

Syntax: **DEIi** **src,** **dest** DEIB

		DEIW
gen	gen	DEIW
read.i	rmw.2i	DEID



The DEIi instruction divides the entire dest operand by the src operand and places the quotient and the remainder in the dest operand location.

The instruction places the quotient in the high-order half of dest and the remainder in the low-order half. The dest operand may be specified as an even-odd General Purpose register pair. In such cases, the instruction places the remainder in the even register and the quotient in the next consecutive (odd) register. The register pair must be specified in assembly language by the name of the even register of the pair.

The src and dest operands are interpreted as unsigned integers.

Flags Affected: None.

Traps: DVZ (Divide by Zero) activated if src equals zero.

Example:

```
DEIW  R2, R0                          CE 2D 10
```

This example divides the double-word value contained in the low-order words of R0 and R1 by the low-order word of register R2. The result is a double-word containing a quotient and a remainder. The remainder is written to the low-order word of register R0; the quotient is written to the low-order word of register R1. The high-order words of registers R0 and R1 are not used or affected.

The instruction is illustrated below:

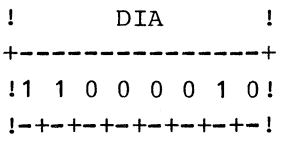
Operands	Operand Values: Hex (Dec) Before	After
R2	AAAA0001 (+1)	AAAA0001 (+1)
R0	BBBBFFFF	BBBB0000
R1	CCCC0000 (+65535)	CCCCFFFF (+65535, rem. 0)

The above case divides 65535 by 1 (H'0000FFFF by H'0001). The quotient is 65535 (in R1), and the remainder is 0 (in R0).

DIA

Diagnose

Syntax: **DIA**



The DIA instruction is intended to support breakpointing circuitry, and is not intended for use in a program. It is a 1-byte instruction which performs a branch to itself, establishing an "infinite loop" which is interruptible. When the loop thus established is interrupted, the return address pushed onto the Interrupt Stack is the address of the DIA instruction itself.

Flag Affected: None.

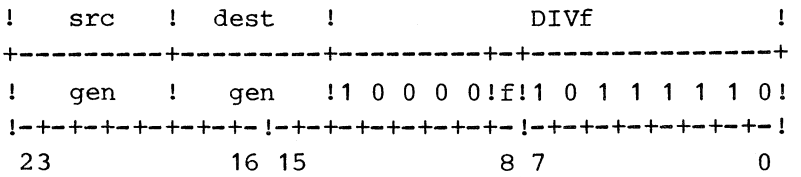
Traps: None.

Example:

```
DIA                                C2
```

Divide Floating

Syntax: **DIVf** **src,** **dest** DIVF
gen gen DIVL
read.f rmw.f



The DIVf instruction divides the dest operand by the src operand and places the result in the dest operand location.

Results for normalized and zero operands are given in the table below. The symbols "+n" and "-n" represent non-zero normalized numbers, positive and negative, respectively. The symbols "+z" and "-z" represent positive and negative zero, respectively.

dest: / src	+n	-n	+z	-z
+n	*	*	+z	-z
-n	*	*	-z	+z

* The result in these cases is the quotient of the two operands.

Flags Affected: No PSR flags. FSR flags are affected as follows:
UF is set if an underflow occurs; unaffected otherwise.
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant to floating-point division are the Divide by Zero exception, caused by attempting to divide a non-zero number by zero, and the Invalid Operation exception, caused by attempting to divide zero by zero.

Examples:

```
1.  DIVF   F0, F7                  BE E1 01
2.  DIVL   -8(FP), 16(SB)          BE A0 C6 78 10
```

Example 1 divides the single-precision number in register F7 by the number in register F0 and places the result in F7.

Example 2 divides the double-precision number at address 16(SB) by the number at address -8(FP) and places the result at address 16(SB).

The instructions are illustrated below:

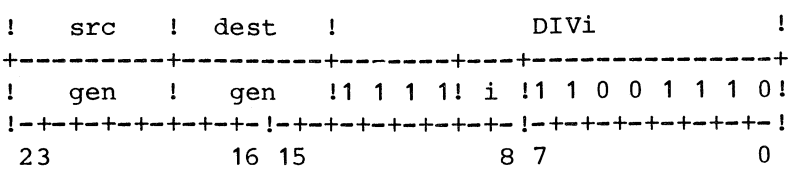
	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	42250000 (+41.25)	42250000 (+41.25)
	F7	434E4000 (+206.25)	40A00000 (+5.0)
Ex. 2:	-8(FP)	409F440000000000 (+2001.0)	409F440000000000 (+2001.0)
	16(SB)	41A2B128DDC00000 (+156800110.875)	40F3218E00000000 (78360.875)

Divide

Syntax: **DIVi src, dest**

gen	gen
read.i	rmw.i

DIVB
DIVW
DIVD



The DIVi instruction divides the dest operand by the src operand, rounds the quotient to the next lower (or more negative) integer, and places the result in the dest operand location. The src and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: DVZ (Divide by Zero) activated if src equals zero.

Examples:

1.	DIVW 10(SP), 4(SP)	CE 7D CE 0A 04
2.	DIVD -6(FP), 12(SB)	CE BF C6 7A 0C

Example 1 divides the word at the address specified by 4(SP) by the word at the address specified by 10(SP). The instruction rounds the quotient and places the result at 4(SP).

Example 2 divides the double-word at the address specified by 12(SB) by the double-word at the address specified by -6(FP). The instruction rounds the quotient and places the result in the double-word at address 12(SB).

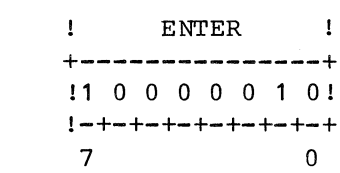
These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	10(SP)	000A (+10)	000A (+10)
	4(SP)	006F (+111)	000B (+11)
Ex. 2:	-6(FP)	00000014 (+20)	00000014 (+20)
	12(SB)	FFFFFF9F (-97)	FFFFFFFB (-5)

In example 1, 111 divided by 10 is 11.1. The next lower integer is 11.
In example 2, -97 divided by 20 is -4.85. The next lower integer is -5.

ENTER
Enter New Procedure Context

Syntax: **ENTER** **reglist,** **constant**
imm disp



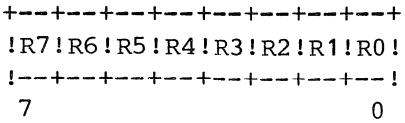
The ENTER instruction creates a "Frame" on the current stack for use by a procedure. A Frame is a block of memory on the stack that provides local storage for the current procedure. The constant operand specifies the number of bytes to be reserved on the stack for local data storage. The Frame Pointer (FP) register is saved and then set up as a pointer from which frame information can be located.

The instruction does the following:

1. Pushes the contents of the FP register (32 bits) onto the stack.
2. Copies the contents of the current stack pointer to the FP register.
3. Subtracts the constant operand from the value of the current stack pointer, lengthening the stack by that number of bytes.
4. Pushes the General-Purpose registers specified by reglist onto the stack.

The reglist operand is specified in assembly language by a list of zero or more General-Purpose register names, enclosed in brackets "[]". The instruction pushes the contents of each register in the list as a double-word onto the currently-selected stack. Register names may appear in any order within reglist but must be separated by commas. Brackets are required even if no register names are given.

In the machine instruction, the reglist operand is encoded in an 8-bit field as shown. Each bit in the field corresponds to one general-purpose register. When the instruction is executed, the instruction reads the bits in the field from right to left beginning with bit 0. If a bit is "0", the instruction ignores the corresponding register. If a bit is "1", it saves the corresponding register.



Flags Affected: None.

Traps: None.

Enter New Procedure Context (continued)

Example:

```
ENTER  [R0, R2, R7], 16          82 85 10
```

This instruction creates a frame on the stack consisting of 16 bytes for local data storage and the contents of register R0, R2, and R7.

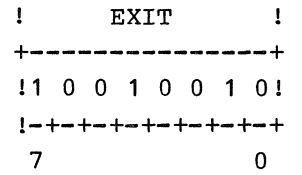
In the following illustration, the PSR S flag is assumed to be 1, selecting SP1 as the current stack pointer.

Operands	Operand Values: Hex Before	After
R0	00000010	00000010
R2	FFFFFFEF	FFFFFFEF
R7	FFFFF9AB	FFFFF9AB
16 (disp)	10 (+16)	--
FP	000010F8	000010EC
SP1	000010F0	000010D0
Stack:		
000010D0	xxxxxxxx	FFFFF9AB
000010D4	xxxxxxxx	FFFFFFEF
000010D8	xxxxxxxx	00000010
000010DC	xxxxxxxx	xxxxxxxx *
000010E0	xxxxxxxx	xxxxxxxx
000010E4	xxxxxxxx	xxxxxxxx
000010E8	xxxxxxxx	xxxxxxxx
000010EC	xxxxxxxx	000010F8
000010F0	AAAAAAAA	AAAAAAAA

* 16 bytes of uninitialized local data storage.

EXIT
Exit Procedure Context

Syntax: **EXIT** **reglist**
imm



The EXIT instruction removes the frame of the current procedure from the stack, restores the former contents of the specified General-Purpose registers (i.e., their contents prior to entering the current procedure), and restores the frame of the previous procedure as the current procedure context.

The instruction does the following:

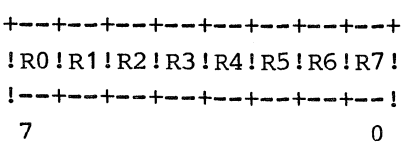
1. Restores the General-Purpose registers specified by reglist by popping them from the current stack.

2. Copies the contents of the FP register to the current stack pointer.

3. Pops the old frame address (32 bits) from the stack to the FP register.

In assembly language, the reglist operand is specified as a list of zero or more General-Purpose register names, enclosed in brackets "[]". The instruction copies to each register in the list a double-word popped from the stack. Register names may appear in any order within reglist but must be separated by commas. Brackets are required even if no register names are given.

In the machine instruction, the reglist operand is encoded in an eight-bit field as shown below. Each bit in the field corresponds to one general-purpose register. When the instruction is executed, the instruction reads the bits in the field from right to left beginning with bit 0. If a bit is "0", the instruction ignores the corresponding register. If a bit is "1", it restores the corresponding register from the stack. Note that the format of the reglist operand is reversed from its format in the ENTER instruction; i.e. bit 0 corresponds to register R7 instead of R0.



Flags Affected: None.

Traps: None.

Exit Procedure Context (continued)

Example:

```
EXIT  [R0, R2, R7]                92 A1
```

This instruction restores the contents of the listed General-Purpose registers, reclaims the frame of the current procedure, and restores the frame of the previous procedure as the current context.

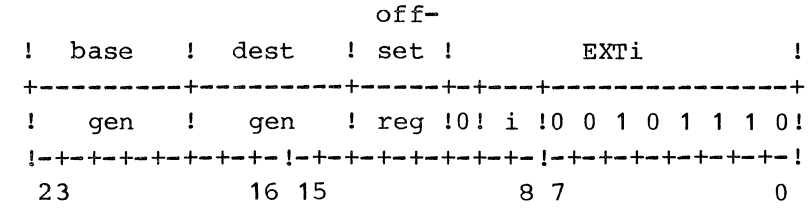
Operands	Operand Values: Hex Before	After
R0	CCCCCCCC	00000010
R2	CCCCCCCC	FFFFFFEF
R7	CCCCCCCC	FFFFF9AB
FP	000010EC	00001000
SP	000010D0	000010F0
Stack:		
000010D0	FFFFF9AB	xxxxxxxx *
000010D4	FFFFFFEF	xxxxxxxx *
000010D8	00000010	xxxxxxxx *
000010DC	BBBBBBBB	xxxxxxxx *
000010E0	BBBBBBBB	xxxxxxxx *
000010E4	BBBBBBBB	xxxxxxxx *
000010E8	BBBBBBBB	xxxxxxxx *
000010EC	00001000	xxxxxxxx *
000010F0	AAAAAAAA	AAAAAAAA

* The EXIT instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

EXTi

Extract Field

Syntax:	**EXTi**	**offset,**	**base,**	**dest,**	**length**	EXTB
		reg	gen	gen	disp	EXTW
			regaddr	write.i		EXTD



The EXTi instruction copies the bit field specified by base, offset and length to the dest operand location. The field is right-justified in dest. High-order bits are zero-filled if the field is shorter than dest or discarded if the field is longer than dest.

The location of the field is taken from the position of its least-significant bit, given by offset and base as follows:

If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.6 for definitions of the operators MOD and DIV above.

Offset is interpreted as a 32-bit signed integer.

Length specifies the number of bits in the field. It must be in the range 1 through 32.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Extract Field (continued)

Example:

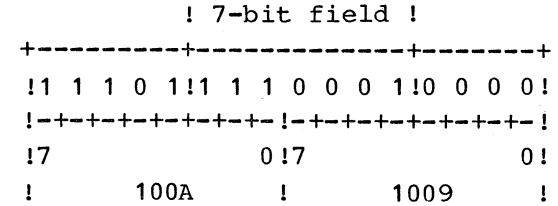
```
EXTW  R0, 0(R1), R2, 7          2E 81 40 00 07
```

This example copies a 7-bit field from memory into the low-order word of register R2. Bits 7 through 15 of register R2 are set to zero and the remaining bits of R2 are unaffected. For designating the location of the field, register R0 supplies the bit offset, and 0(R1) is specified as the base address.

Operands	Operand Values: Hex (Dec) Before	After
R0 (offset)	0000004C (+76)	0000004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 0(R1)	00001000 (+4096)	--
R2	AAAAAAAA	AAAA0071
00001009 * (+4105)	EF10	EF10 **

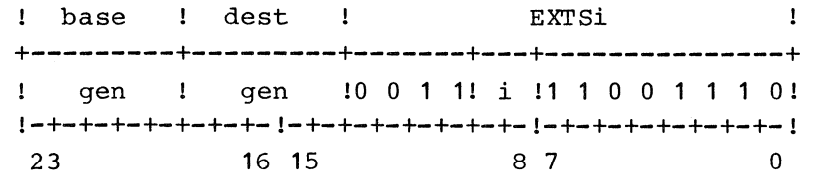
* The address 1009 (Hex) is the effective address of the byte containing the least-significant bit of the specified field. This address is computed as 4096 + (76 DIV 8) = 4105, where 4096 is the base address specified by 0(R1) and 76 is the bit offset given by the contents of register R0.

** The bit field starts at bit position 4 (= 76 MOD 8) in the byte at address 1009 (Hex) and is seven bits long as illustrated.



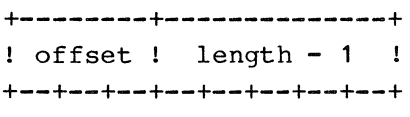
EXTSi
Extract Field Short

Syntax: **EXTSi** **base,** **dest,** **offset, length** EXTSB
gen gen !------imm-----! EXTSW
regaddr write.i EXTSD



The EXTSi instruction copies the bit field specified by base, offset, and length to the dest operand location. The field is right-justified in dest. High-order bits are zero-filled if the field is shorter than dest or discarded if the field is longer than dest.

The offset and length operands are encoded together as an immediate byte appended to the basic instruction. The offset is encoded as the high-order three bits of this byte; the length operand, minus one, is encoded as the low-order five bits. The byte has the following form:



The offset value must be in the range 0 through 7. The length value specifies the number of bits in the field. It must be in the range 1 through 32.

The location of the field is taken from the position of its least-significant bit. If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at the bit position given by offset within the memory byte whose address is given as base.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Extract Field Short (continued)

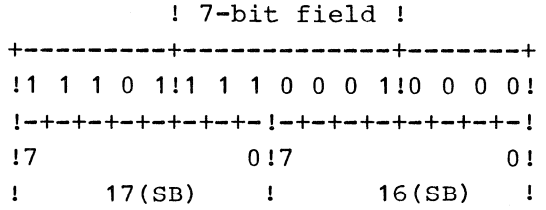
Example:

```
EXTSW  16(SB), R2, 4, 7          CE 8D D0 10 86
```

This example copies a bit field to the low-order word of register R2. The field begins at bit position 4 of the byte at the address specified as 16(SB) and is seven bits long.

Operands	Operand Values: Hex Before	After
R2	AAAAAAAA	AAAA0071 **
16(SB)	EF10	EF10 *

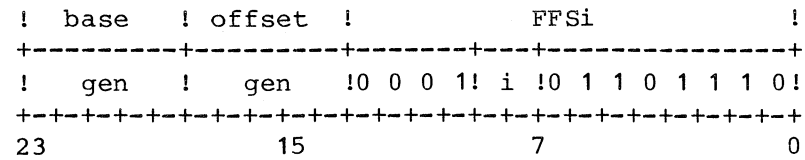
* The bit field starts at bit number 4 in the byte at address 16(SB) and is seven bits long as illustrated:



** The bit field is right-justified in the low-order word of register R2. Nine leading zero bits are added to the bit field to fill the low-order word.

FFSi
Find First Set Bit

Syntax: **FFSi** **base,** **offset** FFSB
gen gen FFSW
read.i rmw.B FFSD



The FFSi instruction searches for the first "1" bit in the base operand. The search starts at the bit specified by the offset operand and proceeds in ascending order to the first "1" bit or to the last bit in base.

If a "1" bit is found, the instruction sets the offset operand value to the bit number of the first "1" bit found in the base operand and clears the F flag in the PSR.

If a "1" bit is not found, the instruction sets the offset operand value to zero and sets the F flag in the PSR to 1.

The offset is interpreted as an unsigned number. Its value must be within the range 0 to 7 (in FFSB instruction), 0 to 15 (in FFSW instruction), and 0 to 31 (in FFSD instruction), otherwise the result placed in the offset operand and in the F bit is undefined.

Note: If the FFSi instruction finds a "1" bit and it is desired to scan the remaining portion of the base operand, the offset operand must be incremented first or the "1" bit previously found must be cleared. Otherwise, the FFSi instruction will detect that bit again.

Flags Affected: F is set if a "1" bit is not found, cleared if found.

Traps: None.

Examples:

1.	FFSW 8(SB), R0	6E 05 D0 08
2.	FFSB -4(FP), TOS	6E C4 C5 7C

Example 1 searches the word at the address specified by 8(SB) for the first "1" bit. The search begins at the bit specified by the low-order byte of register R0, and the result is placed in the low-order byte of R0. The remaining portion of R0 is not used or affected.

Example 2 searches the byte at the address specified by -4(FP). The search begins at the bit specified by the byte on the top of the stack, which is replaced by the resulting bit number.

Find First Set Bit (continued)

These instructions are illustrated below:

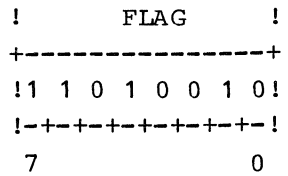
	Operands	Operand Values: Hex (Dec) [Binary] Before	After
Ex. 1:	R0	AAAAAA05 (+5)	AAAAAA08 (+8)
	8(SB)	EF10 [1110111100010000]	EF10 [1110111100010000]
	UPSR	nzfxxltc	nz0xxltc
Ex. 2:	SP	0000FFDE	0000FFDE
	Stack: 0000FFDE	05 (+5)	00 (0)
	-4(FP)	10 [00010000]	10 [00010000]
	UPSR	nzfxxltc	nz1xxltc

In Example 1, the instruction finds the first "1" bit at bit position 8.

In Example 2, the instruction finds no "1" bits; that is, the bits at bit positions 5, 6, and 7 are all "0" bits.

FLAG
Trap on Flag

Syntax: **FLAG**



The FLAG instruction activates the Flag Trap (FLG) if the F flag in the PSR is set. The Flag Trap passes control to the Flag service procedure (see Chapter 6). The return address pushed on the Interrupt Stack is the address of the FLAG instruction itself (see Chapter 6). If the F flag is not set, program execution continues with the next sequential instruction.

Flags Affected: None.

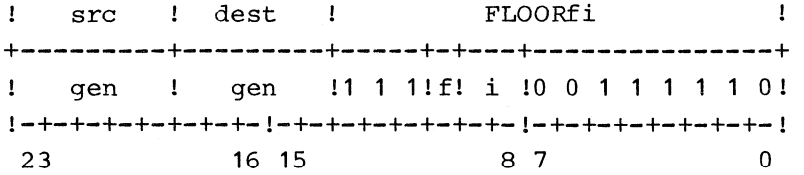
Traps: The Flag Trap (FLG) is activated if the F flag is set.

Example:

```
FLAG                    D2
```

Floor Floating to Integer

Syntax: **FLOORfi** **src,** **dest** FLOORFB FLOORLB
gen gen FLOORFW FLOORLW
read.f write.i FLOORFD FLOORLD



The FLOORfi instruction rounds the src operand to the nearest integer less than, or equal to, it (i.e., toward negative infinity) and places the result in the dest operand location as a signed integer.

Flags Affected: No PSR flags. FSR flags are affected as follows:
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant to this instruction is the Overflow exception, which is caused by attempting to convert a floating-point number that is too great in absolute value to be held in a signed integer of the size specified for dest.

Examples:

```
1.  FLOORFB  F0, R0              3E 3C 00
2.  FLOORLD  F2, 16(SB)          3E BB 16 10
```

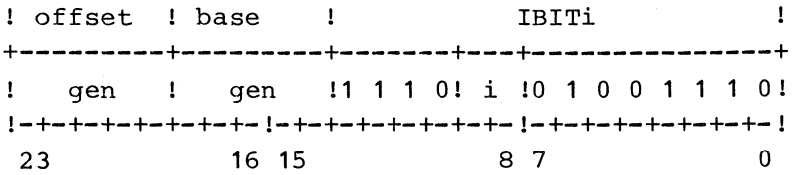
Example 1 rounds the single-precision number in register F0 to a byte-long integer and copies the integer to the low-order byte of register R0.

Example 2 rounds the double-precision number in register pair (F2,F3) to a double-word integer and copies the integer to address 16(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	C0280000 (-2.65)	C0280000 (-2.65)
	R0	AAAAAAAA	AAAAAAFD (-3)
Ex. 2:	(F2,F3)	41C0200888700000 (+541069584.875)	41C0200888700000 (+541069584.875)
	16(SB)	AAAAAAAA	20401110 (+541069584)

Invert Bit

Syntax: **IBITi** **offset,** **base** IBITB
gen gen IBITW
read.i regaddr IBITD



The IBITi instruction inverts (complements) the register or memory bit specified by base and offset after copying the bit value to the F flag in the PSR.

The location of the bit is determined from offset and base. Offset is a general operand, whose length is given by the operation length suffix. Base is an addressing expression giving a byte address from which offset specifies a bit position. See Section 3.5 for details of specifying bit positions.

If base is a register, then the bit is within that register, at the bit position given by the offset operand. If base is a memory location, then the bit is at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.5 for definitions of the operators MOD and DIV above, and for further details of bit instructions.

Offset is interpreted as a signed integer.

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

Traps: None.

Example:

```
IBITW  R0, 1(R1)                    4E 79 02 01
```

This example inverts a bit in memory after copying the bit value into the F flag. For designating the location of the target bit, the low-order word of register R0 supplies the bit offset, and 1(R1) is specified as the base address.

In the following illustration, the target bit is assumed to be 0 prior to instruction execution.

Operands	Operand Values: Hex (Dec) [Binary] Before	After
R0 (offset)	AAAA004C (+76)	AAAA004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 1(R1)	00001001 (+4097)	--
0000100A * (+4106)	EF [11101111]	FF [11111111]
UPSR	nzfxxltc	nz0xxltc

* The address 100A (Hex) is the effective address of the byte containing the desired bit. This address is computed from the offset and the base address as follows:

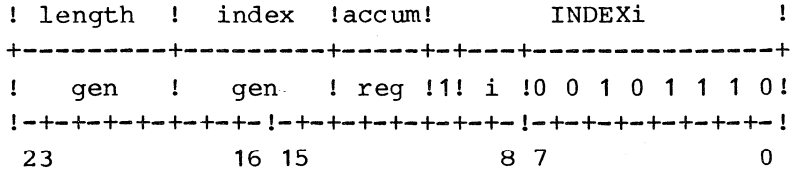
```
base address + (offset DIV 8)
   4097      +       9
   4106, or 100A (Hex)  .
```

The bit number within this byte is calculated as:

```
offset MOD 8
  76   MOD 8
     4  .
```

Calculate Index

Syntax: **INDEXi** **accum,** **length,** **index** INDEXB
reg gen gen INDEXW
read.i read.i INDEXD



The INDEXi instruction assists the programmer in accessing multidimensional arrays by providing a 1-dimensional index which can subsequently be used directly in an addressing mode with Scaled Indexing. The 1-dimensional index is calculated from the values of the indices along each dimension of the array.

This instruction is intended to be executed iteratively, as discussed in Section 3.9, once for each dimension except the first. Each iteration accumulates its result into the general-purpose register specified as accum. The length operand defines the length of the current dimension, giving the difference between the upper and lower index bounds (this is the actual dimension length minus one). The index operand is the zero-adjusted value along the current dimension. The result placed in the accum register is:

accum * (length + 1) + index .

The length and index operands are interpreted as unsigned integers, and are zero-extended to 32 bits internally before use. The accum operand is interpreted as an unsigned 32-bit integer.

Flags Affected: None.

Traps: None.

Example:

```
INDEXB  R0, 20(SB), -4(FP)          2E 04 D6 14 7C
```

This example performs one step of an index calculation. R0 is the accum operand, memory location 20(SB) holds a byte defining the length of the current array dimension, and memory location -4(FP) holds the index value along this dimension.

The case below shows the application of the above instruction to calculate the 1-dimensional index of array element A[I,J], where A has been declared (in the Pascal language) as being of dimensions [1..7, 0..16]. The array is assumed to be stored in row major order (Section 3.9). Since it is an array of only two dimensions, one INDEXi instruction serves to calculate the one-dimensional index.

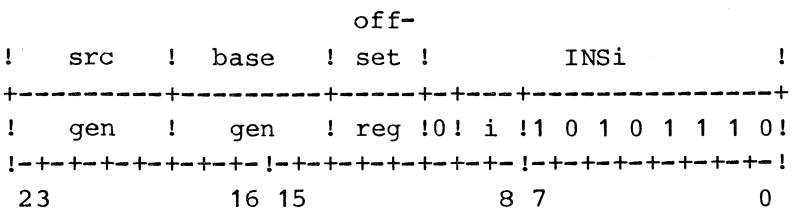
The value of index I (assumed to be 4) has been zero-adjusted to 3 by a CHECK instruction (q.v.), and the result placed in register R0 as a double-word. The value of index J, held in one byte at address -4(FP), is assumed to be 3. The byte at location 20(SB) holds the length operand for the second dimension of the array (16 - 0 = 16).

The result in R0, 54, is the final 1-dimensional index of element [4,3] of array A. This value can be used directly in any addressing mode with a Scaled Indexing modifier to access this array element.

	Operand Values: Hex (Dec)	
Operands	Before	After
R0	00000003 (+3)	00000036 (+54)
20(SB)	10 (+16)	10 (+16)
-4(FP)	03 (+3)	03 (+3)

Insert Field

Syntax: **INSi** **offset,** **src,** **base,** **length** INSB
reg gen gen disp INSW
read.i regaddr INSD



The INSi instruction inserts the src operand into the bit field specified by base, offset, and length. The src operand is right-justified in the field. High-order bits are zero-filled if src is shorter than the field or discarded if src is longer than the field.

The location of the field is taken as the position of its least-significant bit, given by offset and base as follows:

If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.6 for definitions of the operators MOD and DIV above.

Offset is interpreted as a 32-bit signed integer.

Length specifies the number of bits in the field. It must be in the range 1 through 32.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Example:

```
INSW  R0, R2, 0(R1), 7          AE 41 12 00 07
```

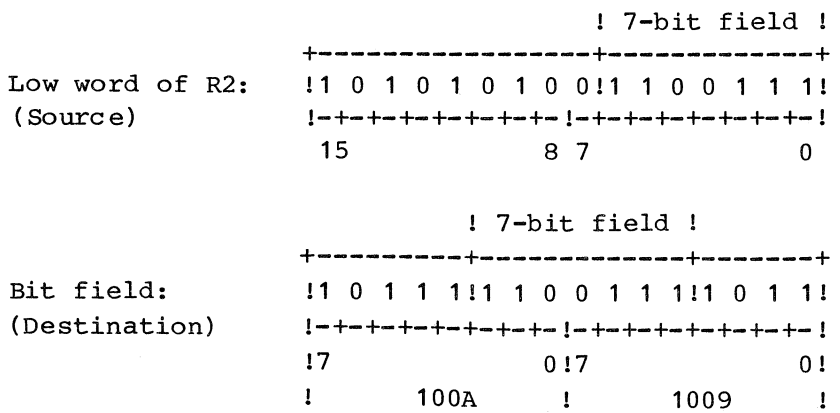
This example inserts seven bits from the low-order word of register R2 into a bit field in memory. For specifying the location of the field, register R0 supplies the bit offset, and 0(R1) is specified as the base address.

The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R0 (offset)	0000004C (+76)	0000004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 0(R1)	00001000 (+4096)	00001000 (+4096)
R2	AAAAAA67	AAAAAA67
00001009 * (+4105)	BBBB	BE7B **

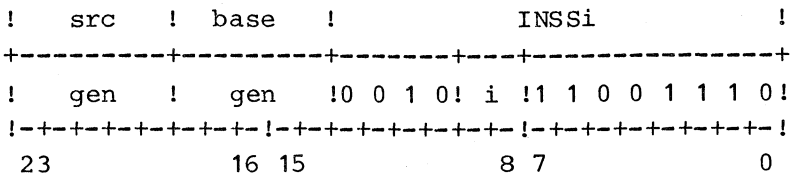
* The address 1009 (Hex) is the effective address of the byte containing the least-significant bit of the specified field. This address is computed as 4096 + (76 DIV 8) = 4105, where 4096 is the address specified by 0(R1) and 76 is the bit offset given by the contents of register R0.

** The bit field starts at bit position 4 (= 76 MOD 8) in the byte at address 1009 (Hex) and is seven bits long as illustrated:



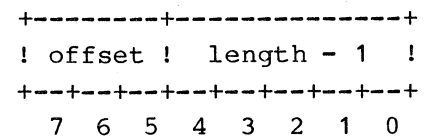
Insert Field Short

Syntax: **INSSi src, base offset, length** INSSB
gen gen !-----imm------! INSSW
read.i regaddr INSSD



The INSSi instruction inserts the src operand into the bit field specified by base, offset, and length. The src operand is right-justified in the field. High-order bits are zero-filled if src is shorter than the field or discarded if src is longer than the field.

The offset and length operands are encoded together as an immediate byte appended to the basic instruction. The offset is encoded as the high-order three bits of this byte; the length operand, minus one, is encoded as the low-order five bits. The byte has the following form:



The offset value must be in the range 0 through 7. The length value specifies the number of bits in the field. It must be in the range 1 through 32.

The location of the field is taken from the position of its least-significant bit. If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at the bit position given by offset within the memory byte whose address is given as base.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Example:

```
INSSW  R2, 16(SB), 4, 7          CE 89 16 10 86
```

This example inserts seven bits from the low-order word of register R2 into a bit field in memory. The bit field begins at bit position 4 in the byte at the address specified by 16(SB) and is seven bits long.

The instruction is illustrated below:

	Operand Values: Hex	
Operands	Before	After
R2	AAAAAA67	AAAAAA67
16(SB)	BBBB	BE7B *

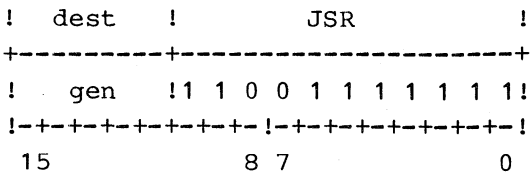
* The bit field starts at bit number 4 in the byte at address 16(SB) and is seven bits long as illustrated:

```
                                    ! 7-bit field !
                   +-----------------+-------------+
Low word of R2:    !1 0 1 0 1 0 1 0 0!1 1 0 0 1 1 1!
(Source)           !-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
                    15              8 7             0

                             ! 7-bit field !
                   +---------+-------------+-------+
Bit field:         !1 0 1 1 1!1 1 0 0 1 1 1!1 0 1 1!
(Destination)      !-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
                   !7              0!7             0!
                   !     17(SB)    !     16(SB)    !
```

Jump to Subroutine

Syntax: **JSR** **dest**
gen
addr



The JSR instruction jumps to the procedure at the address specified by dest after saving the return address on the stack. The return address is the address of the next sequential instruction.

Flags Affected: None.

Traps: None.

Example:

```
JSR 0(4(SB))                    7F 96 04 00
```

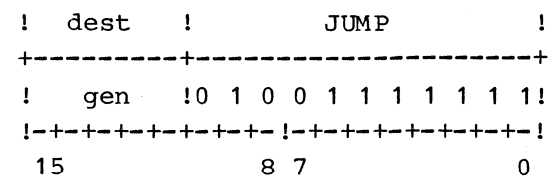
This example causes the program to jump to a procedure at the address held within a double-word at address 4(SB). This is accomplished via the Static Memory Relative addressing mode. The instruction saves the address of the next sequential instruction on the stack.

The instruction is illustrated below:

Operand	Operand Values: Hex Before	After
PC	00009000	00001FFF
4(SB)	00001FFF	00001FFF
SP	0000FFD4	0000FFD0
Stack:		
0000FFD0	xxxxxxxx	00009004
0000FFD4	AAAAAAAA	AAAAAAAA

JUMP
Jump

Syntax: **JUMP** **dest**
gen
addr



The JUMP instruction jumps to the address specified by dest by loading the effective address of dest into the PC register.

Flags Affected: None.

Traps: None.

Example:

```
JUMP 0(-8(FP))                          7F 82 78 00
```

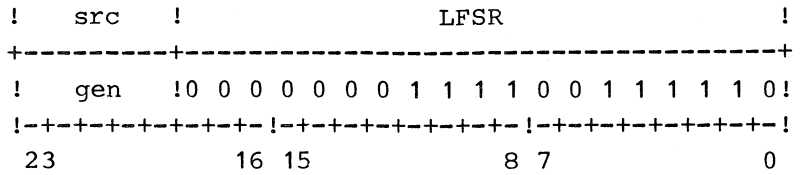
This example loads the address held in the double-word at address -8(FP) into the PC register. This is accomplished via the Frame Memory Relative addressing mode. Program execution continues at that address.

The instruction is illustrated below:

	Operand Values: Hex	
Operand	Before	After
-8(FP)	00001004	00001004
PC	0000909A	00001004

Load Floating-Point Status Register (FSR)

Syntax: **LFSR** **src**
gen
read.D



The LFSR instruction copies the double-word specified by src to the Floating Point Status register (FSR). See Section 2.4.2 for the format of the FSR.

Flags Affected: No PSR flags. All FSR flags are affected.
All implemented FSR fields are loaded from the arc operand.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Example:

```
LFSR  R0                          3E 0F 00
```

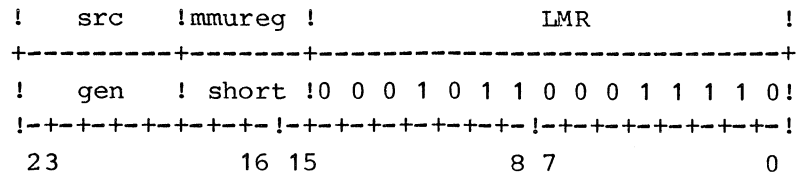
This example copies the contents of register R0 into the FSR.

	Operand Values: Hex	
Operands	Before	After
R0	00000028	00000028
FSR	xxxx0129	xxxx0028

LMR

Load Memory Management Register

Syntax: **LMR** **mmureg, src**
short gen
read.D



The LMR instruction copies the src operand to the Memory Management register specified by mmureg.

The LMR instruction may load the following registers. The short field of the basic instruction holds a 4-bit value which addresses the corresponding Memory Management register as shown below.

Register	mmureg	short field
Breakpoint Register 0	BPR0	0000
Breakpoint Register 1	BPR1	0001
Program Flow Register 0	PF0	0100
Program Flow Register 1	PF1	0101
Sequential Count Registers	SC	1000
Memory Management Status Register	MSR	1010
Breakpoint Count Register	BCNT	1011
Page Table Base Register 0	PTB0	1100
Page Table Base Register 1	PTB1	1101
Error/Invalidate Address Register	EIA	1111

Flags Affected: None.

Traps: Undefined Instruction Trap (UND) is activated if the M bit in the CFG register is clear. The instruction is not executed.

Illegal Instruction Trap (ILL) is activated if the U flag is set. The instruction is not executed.

Load Memory Management Register (continued)

Example:

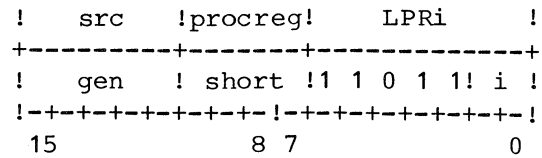
```
LMR  BCNT, R0                    1E 8B 05
```

This example copies the contents of register R0 to the Breakpoint Count Register.

Operands	Operand Values: Hex Before	After
R0	00009000	00009000
BCNT	xxAAAAAA	xx009000

LPRi
Load Processor Register

Syntax: **LPRi** **procreg, src** LPRB
short gen LPRW
read.i LPRD



The LPRi instruction copies the src operand to the dedicated register specified by procreg. See Section 2.2 for the formats of these registers.

The src operand value is right-justified in the register. In registers other than PSR, the high-order bits are zero-filled if src is shorter than the register. The "LPRB PSR" form loads only the low-order byte of the PSR, but is privileged (see note 3 below). High-order src bits are discarded if src is longer than the register.

The Load Processor Register instruction may load the following registers. The specified procreg corresponds to the 4-bit short field in the basic instruction as shown below.

Register	procreg	short	
User PSR	UPSR	0000	(Note 3)
Frame Pointer	FP	1000	
Stack Pointer	SP	1001	(Note 1)
Static Base Register	SB	1010	(Note 2)
Processor Status Register	PSR	1101	(Note 3)
Interrupt Base Register	INTBASE	1110	(Note 3)
Module Register	MOD	1111	

NOTES:
1. If SP is specified in the instruction and the S flag in the PSR is set, the instruction copies the src operand to the SP1 register. If the S flag is clear, the instruction copies the operand to the SP0 register.

2. The SB register should not be loaded using LPRi except during system initialization after a Reset, because it is automatically reloaded from the current Module Table entry whenever an external procedure returns, or a trap or interrupt service procedure returns.

3. Specifying UPSR as the procreg operand causes only the low-order byte of the PSR to be affected, regardless of the operation length specified, and is not a privileged operation. Specifying PSR or INTBASE as the procreg operand is always a privileged operation, regardless of the operation length. See "Traps" below.

Load Processor Register (continued)

Flags Affected: All PSR flags are affected if PSR is specified with operation length of W or D. The N, Z, F, L, T and C flags are affected when UPSR is selected, or PSR is selected with operation length B. No flags are affected otherwise.

Traps: Illegal Operation Trap (ILL) is activated if the U flag is set and PSR or INTBASE is specified.

Examples:

```
1.  LPRD  FP, R0              6F 04
2.  LPRW  MOD, 4(SB)          ED D7 04
```

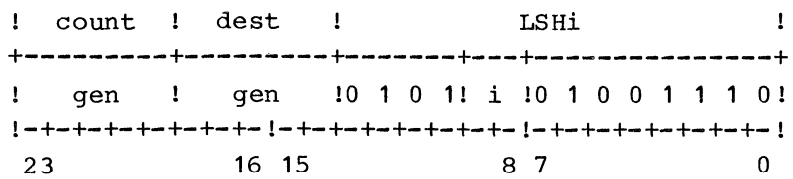
Example 1 loads the entire FP register from register R0.

Example 2 copies the word at address 4(SB) into the MOD register.

	Operands	Operand Values: Hex Before	After
Ex. 1:	FP	00AAAAAA	00543210
	R0	00543210	00543210
Ex. 2:	MOD	AAAA	0030
	4(SB)	0030	0030

LSHi
Logical Shift

Syntax: **LSHi** **count, dest** LSHB
gen gen LSHW
read.B rmw.i LSHD



The LSHi instruction performs a logical shift on the dest operand in the manner specified by the count operand. The sign of count determines the direction of the shift. The absolute value of count gives the number of bit positions to shift the dest operand.

The count operand value must be within the range -7 to +7 for the LSHB form, -15 to +15 for the LSHW form, and -31 to +31 for the LSHD form. A positive count specifies a left shift; a negative count specifies a right shift. In a logical shift, all bits shifted out of dest are lost, and all bit positions emptied by the shift are zero-filled.

The count operand is interpreted as a signed integer. The dest operand is interpreted as an unsigned integer.

Flags Affected: None.

Traps: None.

Logical Shift (continued)

Examples:

```
1.  LSHB   4, 8(SB)              4E 94 A6 04 08
2.  LSHB  -4(FP), 8(SB)          4E 94 C6 7C 08
```

Example 1 shifts the 1-byte operand at address 8(SB) four bit positions to the left.

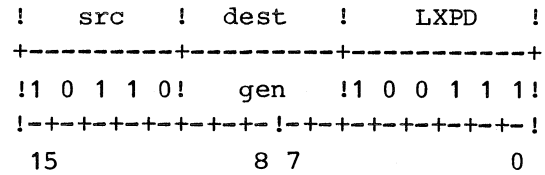
Example 2 shifts the operand at address 8(SB) according to the count given by the byte at address -4(FP). This value, -1, causes a 1-bit logical right shift.

	Operands	Operand Values: Binary (Dec) Before	After
Ex. 1:	4 (immediate)	00000100 (+4)	--
	8(SB)	11111110	11100000
Ex. 2:	-4(FP)	11111111 (-1)	11111111 (-1)
	8(SB)	11111110	01111111

LXPD

Load External Procedure Descriptor

Syntax: **LXPD** **src, dest**
gen gen
addr write.D



The LXPD instruction is a specific form of the ADDR instruction whose effect is to copy an external procedure descriptor (Section 2.7.3) from a specified entry of the current Link Table into the dest operand location. The mnemonic "LXPD" is provided by the NSC Series 32000 assembler as an alternative to "ADDR" when this use is intended. See the Series 32000 Cross-Assembler Reference Manual for further details.

Flags Affected: None.

Traps: None.

Example:

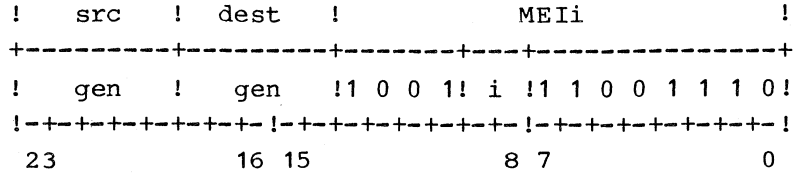
```
LXPD  EXT(3),TOS                    E7 B5 03 00
```

This example copies the external procedure descriptor from Link Table entry number 3, pushing it as a double-word onto the currently-selected stack.

Operands	Operand Values: Hex Before	After
Link Table Entry 3	01000030 (module 0030) (offset 0100)	01000030
SP	0000FFE4	0000FFE0
Stack:		
0000FFE0	xxxxxxxx	01000030
0000FFE4	AAAAAAAA	AAAAAAAA

Multiply Extended Integer

Syntax: **MEIi** **src,** **dest** MEIB
gen gen MEIW
read.i rmw.2i MEID



The MEIi instruction multiplies the src operand and the low-order half of the dest operand and places the result in the entire dest operand location.

The src and dest operands are interpreted as unsigned integers.

The dest operand may be specified as an even-odd General-Purpose register pair. In such cases, the instruction reads the even-numbered register of the pair and places the low-order half (1, 2 or 4 bytes) of the result in the even register and the high-order half in the next consecutive register. The register pair must be specified in assembly language by the name of the even register of the pair.

If the Top of Stack (TOS) addressing mode is used for the dest operand, the Stack Pointer contents do not change. Note that this is not the same as popping a value of length "i" and pushing a result of length "2i". Space must already have been allocated on the stack to accommodate the entire result.

Flags Affected: None.

Traps: None.

Examples:

```
1.  MEIW  R2, 10(SB)          CE A5 16 0A
2.  MEIW  R2, R0              CE 25 10
```

Example 1 multiplies the low-order word of register R2 and the word at the dest operand address 10(SB) and places the double-word result at the dest operand address 10(SB).

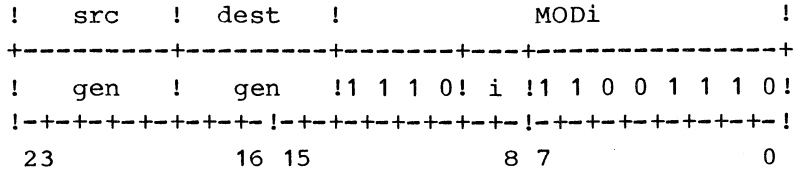
Example 2 multiplies the low-order word of register R2 and the low-order word of register R0. The result is a double-word. The low-order word of the result is written to the low-order word of register R0, the high-order word is written to the low-order word of register R1.

These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R2	AAAA0020 (+32)	AAAA0020 (+32)
	10(SB)	BBBB1001 (+4097)	00020020 (+131104)
Ex. 2:	R2	AAAA0020 (+32)	AAAA0020 (+32)
	R0	BBBB1001	BBBB0020
	R1	CCCCCCCC (+4097)	CCCC0002 (+131104)

Modulus

Syntax: **MODi** **src,** **dest** MODB
gen gen MODW
read.i rmw.i MODD



The MODi instruction places the value dest modulo src in the dest operand location. The modulus is the remainder after division is performed as per the DIVi instruction (q.v.). It is computed as

```
dest - ((dest DIV src) * src)
```

where dest DIV src is rounded to the next integer less than or equal to the exact quotient. The result of a MODi instruction always has the sign of the src operand (i.e., the divisor) unless the result is zero, which is always positive. Compare the REMi instruction (q.v.).

The src and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: Divide by Zero Trap (DVZ) is activated if src equals zero.

Example:

```
MODB  4(SB), 8(SB)                    CE B8 D6 04 08
```

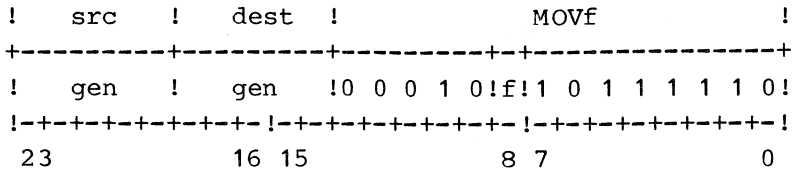
This example computes the modulus of the operands specified by 4(SB) and 8(SB) and places the result in the byte at 8(SB).

The action of this instruction for four different cases is illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Case 1:	4(SB)	0A (+10)	0A (+10)
	8(SB)	1F (+31)	01 (+1)
Case 2:	4(SB)	F6 (-10)	F6 (-10)
	8(SB)	1F (+31)	F7 (-9)
Case 3:	4(SB)	F6 (-10)	F6 (-10)
	8(SB)	E1 (-31)	FF (-1)
Case 4:	4(SB)	0A (+10)	0A (+10)
	8(SB)	E1 (-31)	09 (+9)

Move Floating Point

Syntax: **MOVf** **src,** **dest** MOVF
gen gen MOVL
read.f write.f



The MOVF instruction copies the src operand to the dest operand location.

Flags Affected: No PSR Flags.
The FSR TT field is set to all zeroes.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Example:

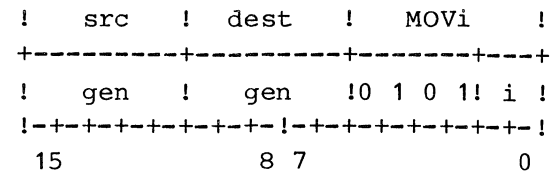
```
MOVF  F0, 8(SB)                    BE 85 06 08
```

This example moves the single-precision number in register F0 to the operand at address 8(SB).

Operands	Operand Value: Hex (Dec) Before	After
F0	3F800000 (+1.0)	3F800000 (+1.0)
8(SB)	AAAAAAAA	3F800000 (+1.0)

MOVi
Move

Syntax: **MOVi** **src,** **dest** MOVB
gen gen MOVW
read.i write.i MOVD



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

Flags Affected: None.

Traps: None.

Example:

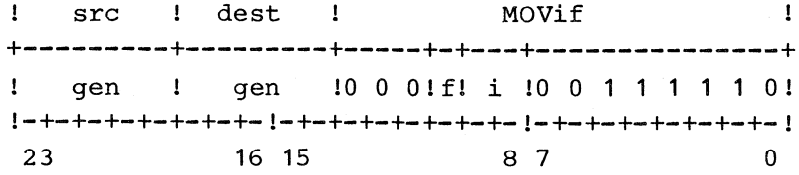
```
MOVD  R0, 8(SB)                    97 06 08
```

This example copies the contents of register R0 to the double-word at the address specified by 8(SB).

Operands	Operand Values: Hex Before	After
R0	12345678	12345678
8(SB)	AAAAAAAA	12345678

Move Converting Integer to Floating Point

Syntax: **MOVif** **src,** **dest** MOVBF MOVBL
gen gen MOVWF MOVWL
read.i write.f MOVDF MOVDL



The MOVif instruction converts the integer operand src to a single- or double-precision floating-point number and places the result in the dest operand location.

Rounding, if required, is controlled by the Rounding Mode bits in the FSR.

Flags Affected: No PSR flags. FSR flags are affected as follows:
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant is the Inexact Result trap if it is enabled in the FSR. It can occur in the MOVDF form, because in this case there are fewer significant bits in dest than in src. The smallest integer values for which this will happen are +16,777,217 (01000001 Hex) and -16,777,217 (FEFFFFFF Hex).

Examples:

```
1.  MOVBF   2, F0                   3E 04 A0 02
2.  MOVDL   16(SB), F2              3E 83 D0 10
```

Example 1 converts the integer constant 2 to a single-precision number and copies the number to the register F0.

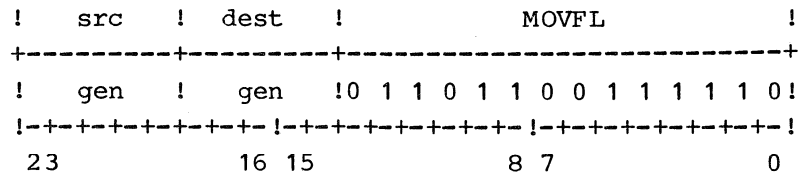
Example 2 converts the double-word integer at the address specified by 16(SB) to a double-precision number and places the number into the register pair (F2,F3).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	2 (immediate)	02 (+2)	--
	F0	AAAAAAAA	40000000 (+2.0)
Ex. 2:	16(SB)	20401110 (+541069584)	20401110 (+541069584)
	(F2,F3)	AAAAAAAAAAAAAAAA	41C0200888000000 (+541069584.0)

Move Floating to Long Floating

Syntax: **MOVFL src, dest**

	src	dest
	gen	gen
	read.F	write.L



The MOVFL instruction converts the src operand to double-precision format and places the result in the dest operand location.

Flags Affected: No PSR Flags.
The FSR TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Example:

```
MOVFL  8(SB), F0                    3E 1B D0 08
```

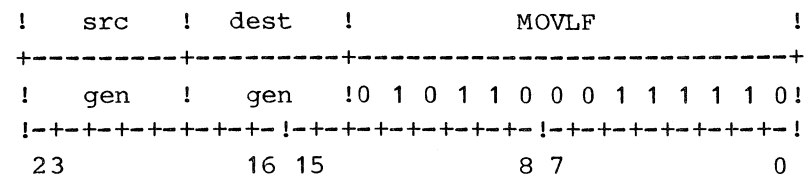
This example converts the single-precision number at the address specified by 8(SB) to a double-precision number and places the number in the register pair (F0,F1).

	Operand Values: Hex (Dec)	
Operands	Before	After
8(SB)	3F800000 (+1.0)	3F800000 (+1.0)
F0	AAAAAAAAAAAAAAAA	3FF0000000000000 (+1.0)

MOVLF

Move Long Floating to Floating

Syntax: **MOVLF** **src,** **dest**
gen gen
read.L write.F



The MOVLF instruction converts the src operand to a single-precision number and places the result in the dest operand location.

Rounding is performed, if necessary, according to the rounding mode selected in the FSR. See Section 3.3 for details of rounding modes.

Flags Affected: No PSR flags. FSR flags are affected as follows:
UF is set if an underflow occurs; unaffected otherwise.
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant cases are:

- Overflow which occurs if the src operand is too great in absolute value to be represented as a single-precision number.
- Underflow which, if enabled in the FSR, occurs if the src operand is too small in absolute value to be represented as a normalized single-precision number.
- Inexact Result which, if enabled in the FSR, occurs if a loss of precision occurs in the conversion.

Move Long Floating to Floating (continued)

Example:

```
MOVLF  F0, 12(SB)                    3E 96 06 0C
```

This example converts the double-precision number in register pair (F0,F1) to a single-precision number and places the result at address 12(SB).

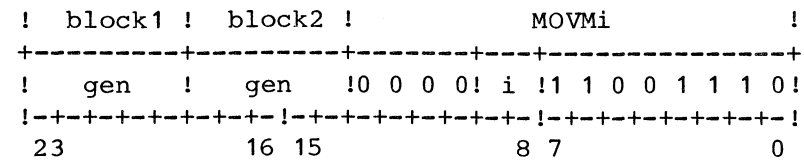
Operands	Operand Values: Hex (Dec) Before	After
F0	3FF0000000000000 (+1.0)	3FF0000000000000 (+1.0)
12(SB)	AAAAAAAA	3F800000 (+1.0)

MOVMi
Move Multiple

Syntax: **MOVMi** **block1,** **block2,** **length**

	gen	gen	disp
	addr	addr	

MOVMB
MOVMW
MOVMD



The MOVMi instruction copies the contents of block1 to block2. The instruction copies consecutive integers from block1 to consecutive integer locations in block2.

In assembly language, the length operand is specified as the number of integers in each block. In the machine instruction, however, the length operand is encoded according to the formula

(num - 1) * i

where num is the number of integers in each block, and i is the number of bytes per integer. A block may not be greater than 16 bytes in length.

Flags Affected: None.

Traps: None.

Example:

```
MOVMW 10(R0), 16(R1), 4          CE 41 42 0A 10 06
```

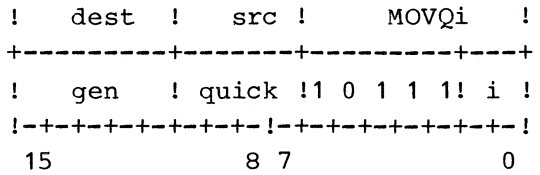
This instruction copies four word-long integers from the block starting at the address specified by 10(R0) to the block starting at the address specified by 16(R1).

Operands	Operand Values: Hex Before	After
R0	00002000	00002000
R1	0000F000	0000F000
0000200A *	1FBE 10A9 8729 6511	1FBE 10A9 8729 6511
0000F010 **	AAAA AAAA AAAA AAAA	1FBE 10A9 8729 6511

* The address of the first block, as specified by 10(R0).
** The address of the second block, as specified by 16(R1).

Move Quick Integer

Syntax: **MOVQi** **src,** **dest** MOVQB
quick gen MOVQW
write.i MOVQD



The MOVQi instruction copies the src operand to the dest operand location. Before the copy operation, src is sign-extended to the length of dest.

Flags Affected: None.

Traps: None.

Example:

```
MOVQW  7, TOS                    DD BB
```

This example pushes the quick value 7 as a word onto the top of the stack. The high-order bits of the result are zero-filled due to sign-extension.

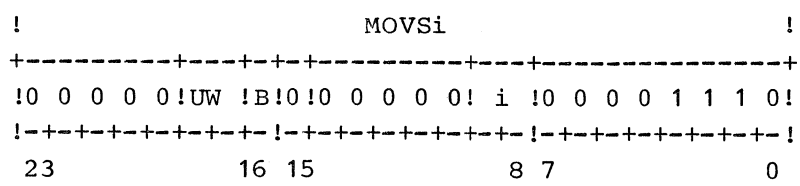
	Operand Values: Hex	
Operands	Before	After
7 (quick)	0007 *	--
SP	0000FFEE	0000FFEC
Stack:		
0000FFEC	xxxx	0007
0000FFEE	AAAA	AAAA

* This value is the internal representation of the Quick value 7, after sign-extension to Word length. The operand is encoded within the instruction as binary 0111.

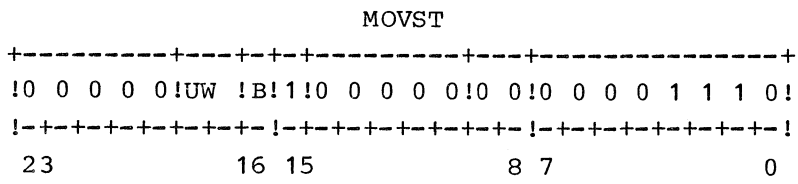
MOVSi
MOVST
Move String

Syntax: **MOVSi** **options**

MOVSB
MOVSW
MOVSD
MOVST



Syntax: **MOVST** **options**



Operands of the MOVSi and MOVST instructions are specified in General-Purpose registers:

R0 - Number of string elements to be processed.
R1 - Address of current String 1 element.
R2 - Address of current String 2 element.
R3 - Address of translation table (MOVST form only).
R4 - Match value (with Until Match or While Match option only).

The MOVSi instruction copies consecutive elements of String 1 (address in R1) to consecutive element locations in String 2 (address in R2). After an element is copied, the instruction sets register R1 to the address of the next element to copy, sets register R2 to the address of the next location to receive an element, and sets R0 to the number of elements remaining to be copied. See Section 3.7 for the exact sequences followed by String instructions.

The MOVST instruction copies one-byte elements from String 1, after translation, to String 2. The translated value to be copied is found by adding the current element from String 1 as an unsigned integer to the translation table address found in register R3. The instruction copies elements and sets registers as described above. See Section 3.7 for details of string translation.

Options may be specified by listing the letters B (Backward), U (Until Match) and W (While Match) as operands. The U and W options are mutually exclusive. See Section 3.7 for details of the options available in String instructions.

Move String (continued)

In the machine instruction, the options are encoded in the B and UW fields as follows:

B field =	0	Forward direction.
	1	Backward direction.
UW field =	00	Neither Until Match nor While Match.
	01	While Match.
	10	(reserved)
	11	Until Match.

String instructions are interruptible. See Section 3.7.

Flags Affected: F is set if the U or W option is specified and the corresponding Until/While condition is met, otherwise it is cleared.

Traps: None.

Example:

```
MOVST                                   0E 80 00
```

This example moves byte-long integers from the first string, after translation, to the second string.

The instruction is as follows:

Operands	Operand Values: Hex (Dec) Before	After
R0	00000020 (+32)	00000000 (0)
R1	00002000	00002021
R2	0000F000	0000F021
R3	00010000	00010000
UPSR	nzfxxltc	nz0xxltc

Translation Table Contents

```
10000    00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
         16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

String Contents Before

```
2000     1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
         1F 1D 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18

F000     AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
         AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
```

String Contents After

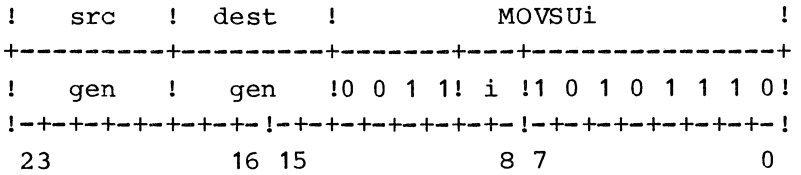
```
2000     1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
         1F 1D 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18

F000     30 04 05 28 10 20 12 11 09 07 31 15 23 01 00 17
         31 29 26 09 01 18 20 14 30 10 00 03 09 06 22 24
```

This example translates 32 binary integers (in the range 0-31) into binary coded decimal (BCD) values. Each integer is read from String 1 (address given by R1) and used as an offset into the translation table at address 10000 (Hex). The BCD value found at that address in the translation table is then copied to the current location in String 2 (address in R2).

Move Value from Supervisor to User Space

Syntax: **MOVSUi** **src,** **dest** MOVSUB
gen gen MOVSUW
addr addr MOVSUD



The MOVSUi instruction copies the src operand in Supervisor space to the dest operand location in User space. User Mode protection is applied to the User space access.

Flags Affected: None.

Traps: Illegal Instruction Trap (ILL) is activated if the U flag is set.

Example:

```
MOVSUB  5(SP), 9(SB)              AE 8C CE 05 09
```

This example copies the byte at the operand address 5(SP) in supervisor space to the operand address 9(SB) in user space.

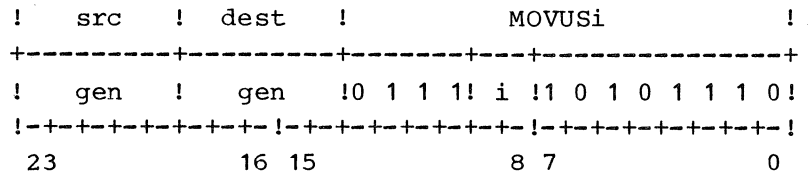
The instruction is illustrated below:

	Operand Values: Hex	
Operands	Before	After
5(SP) Supervisor	10	10
9(SB) User	AA	10

MOVUSi

Move Value from User to Supervisor Space

Syntax: **MOVUSi** **src,** **dest** MOVUSB
gen gen MOVUSW
addr addr MOVUSD



The MOVUSi instruction moves the src operand in User space to the dest operand location in Supervisor space. User Mode protection is applied to the User space access.

Flags Affected: None.

Traps: Illegal Instruction Trap (ILL) is activated if the U flag is set.

Example:

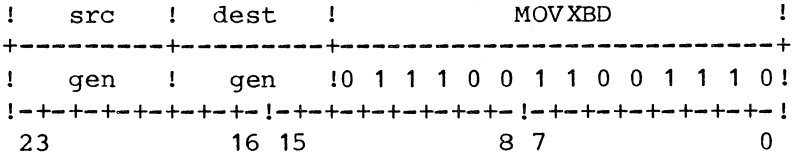
```
MOVUSB  9(SB), 5(SP)              AE 5C D6 09 05
```

This example moves the byte at the address specified by 9(SB) in user space to the address specified by 5(SP) in supervisor space.

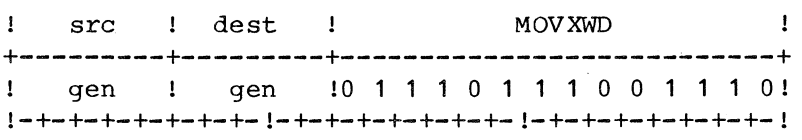
Operands	Operand Values: Hex Before	After
9(SB) User	10	10
5(SP) Supervisor	AA	10

Move with Sign-Extension

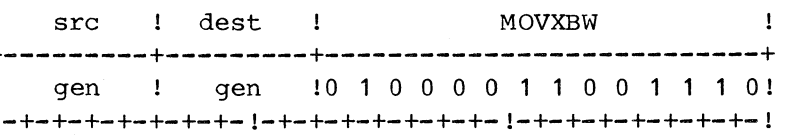
Syntax: **MOVXBD** **src,** **dest** MOVXBD
gen gen MOVXWD
read.B write.D MOVXBW



Syntax: **MOVXWD** **src,** **dest**
gen gen
read.W write.D



Syntax: **MOVXBW** **src,** **dest**
gen gen
read.B write.W



The MOVX instructions convert signed integers to any greater length while preserving their signed values through sign-extension.

The MOVXBD instruction copies the byte-length src operand to the low-order byte of the double-word dest operand and extends the src operand's sign bit through the remaining high-order bits of the dest operand.

The MOVXWD instruction copies the word-length src operand to the low-order word of the double-word dest operand and extends the src operand's sign bit through the remaining high-order bits of the dest operand.

The MOVXBW instruction copies the byte-length src operand to the low-order byte of the word dest operand and extends the src operand's sign bit through the remaining high-order bits of the dest operand.

The src and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: None.

Example:

```
MOVXBW  2(SB), R0                    CE 10 D0 02
```

This example copies the byte at the address specified by 2(SB) to the low-order byte of register R0 and extends the sign bit of the byte through the next eight bits of R0. The instruction affects the low-order word of R0 only.

The instruction (for two cases) is as follows:

	Operands	Operand Values: Hex (Dec) Before	After
Case 1:	2(SB)	F0 (-16)	F0 (-16)
	R0	AAAAAAAA	AAAAFFF0 (-16)
Case 2:	2(SB)	70 (+112)	70 (+112)
	R0	AAAAAAAA	AAAA0070 (+112)

Move with Zero-Extension

Syntax: **MOVZBD** **src,** **dest** MOVZBD
gen gen MOVZWD
read.B write.D MOVZBW

```
!   src   ! dest  !           MOVZBD           !
+---------+-------+----------------------------+
!   gen   !  gen  !0 1 1 0 0 0 1 1 0 0 1 1 1 0!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 23          16 15            8 7             0
```

Syntax: **MOVZWD** **src,** **dest**
gen gen
read.W write.D

```
!   src   ! dest  !           MOVZWD           !
+---------+-------+----------------------------+
!   gen   !  gen  !0 1 1 0 0 1 1 1 0 0 1 1 1 0!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 23          16 15            8 7             0
```

Syntax: **MOVZBW** **src,** **dest**
gen gen
read.B write.W

```
!   src   ! dest  !           MOVZBW           !
+---------+-------+----------------------------+
!   gen   !  gen  !0 1 0 1 0 0 1 1 0 0 1 1 1 0!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 23          16 15            8 7             0
```

The MOVZ instructions convert unsigned integers to any greater length while preserving their unsigned values through zero-extension.

The MOVZBD instruction copies the byte-length src operand to the low-order byte of the double-word dest operand and zero-fills the remaining high-order bits of the dest operand.

The MOVZWD instruction copies the word-length src operand to the low-order word of the double-word dest operand and zero-fills the remaining high-order bits of the dest operand.

The MOVZBW instruction copies the src operand to the low-order byte of the dest operand and zero-fills the remaining eight bits of dest.

The src and dest operands are interpreted as unsigned integers.

Flags Affected: None.

Traps: None.

Example:

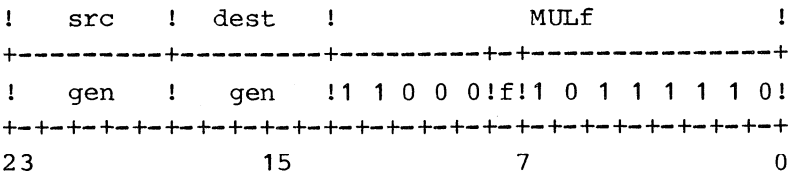
```
MOVZBW  -4(FP), R0                    CE 14 C0 7C
```

This example copies the byte at the address specified by -4(FP) to the low-order byte of register R0 and sets the next eight bits of register R0 to zero. The instruction affects only the low-order word of R0.

Operands	Operand Values: Hex (Unsigned Dec) Before	After
-4(FP)	FF (+255)	FF (+255)
R0	AAAAAAAA	AAAA00FF (+255)

Multiply Floating

Syntax: **MULf** **src,** **dest** MULF
gen gen MULL
read.f rmw.f



The MULf instruction multiplies the src and dest operands and places the result in the dest operand location. Results for normalized and zero operands are given in the table below. The symbols "+n" and "-n" represent non-zero normalized numbers, positive and negative, respectively. The symbols "+z" and "-z" represent positive and negative zero, respectively.

dest: / src	+n	-n	+z	-z
+n	*	*	+z	-z
-n	*	*	-z	+z
+z	+z	-z	+z	-z
-z	-z	+z	-z	+z

* The result in these cases is the product of the two operands.

Flags Affected: No PSR flags. FSR flags are affected as follows:
UF is set if an underflow occurs; unaffected otherwise.
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Examples:

```
1.  MULF  F0, F7                    BE F1 01
2.  MULL  -8(FP), 8(SB)             BE B0 C6 78 08
```

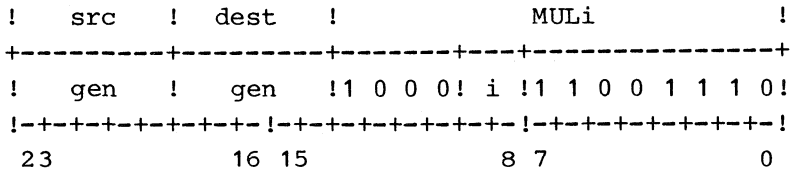
Example 1 multiplies the single-precision numbers in registers F0 and F7 and places the result in register F7.

Example 2 multiplies the double-precision numbers at addresses -8(FP) and 8(SB) and places the double-precision result at address 8(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	42250000 (+41.25)	42250000 (+41.25)
	F7	40A00000 (+5.0)	434E4000 (+206.25)
Ex. 2:	-8(FP)	409F440000000000 (+2001.0)	409F440000000000 (+2001.0)
	8(SB)	40F3218E00000000 (+78360.875)	41A2B128DDC00000 (+156800110.875)

Multiply

Syntax: **MULi** **src,** **dest** MULB
gen gen MULW
read.i rmw.i MULD



The MULi instruction multiplies the src and dest operands and places the product in the dest operand location. If the product is longer than dest, the high-order bits are truncated.

Flags Affected: None.

Traps: None.

Examples:

1. MULW 5, R0 CE 21 A0 00 05
2. MULD 4(-4(FP)), 3(SB) CE A3 86 7C 04 03

Example 1 multiplies the constant 5 and the low-order word of register R0 and places the result in the low-order word of register R0.

Example 2 multiplies the double-word at the memory address specified by 4(-4(FP)) by the double-word at the address specified by 3(SB). The instruction places the result in the double-word at address 3(SB).

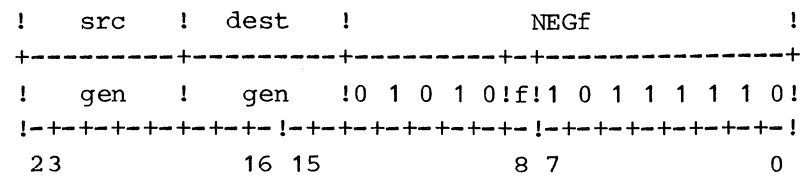
These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	5 (immediate)	0005 (+5)	--
	R0	BBBB000A (+10)	BBBB0032 (+50)
Ex. 2:	4(-4(FP))	FFFFFFFE (-2)	FFFFFFFE (-2)
	3(SB)	FFFFFFF0 (-16)	00000020 (+32)

NEGf

Negate Floating

Syntax: **NEGf** **src,** **dest** NEGF
gen gen NEGL
read.f write.f



The NEGf instruction complements the sign bit of the src operand and places the result in the dest operand location.

Flags Affected: No PSR flags.
The FSR TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Example:

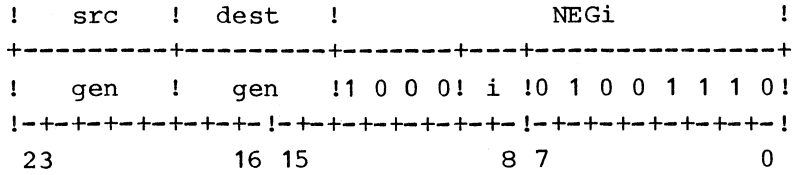
```
NEGF  F0, F2                    BE 95 00
```

This example complements the sign bit of the single-precision number in register F0 and places the result in register F2.

Operands	Operand Values: Hex (Dec) Before	After
F0	42250000 (+41.25)	42250000 (+41.25)
F2	AAAAAAAA	C2250000 (-41.25)

Negate

Syntax: **NEGi** **src,** **dest** NEGB
gen gen NEGW
read.i write.i NEGD



The NEGi instruction negates (takes the two's complement of) the src operand by subtracting src from zero and places the result in the dest operand location.

Flags Affected: C is set on a borrow from subtraction, cleared if no borrow. (A borrow will always occur in this instruction unless the src operand is zero.)

F is set on an overflow from subtraction, cleared if no overflow. This condition will occur if the src operand is the most negative number that can be represented in the operand length specified by the programmer. This value for bytes is -128 (Hex 80); for words it is -32768 (Hex 8000) and for double-words it is -2,147,483,648 (Hex 80000000). These values have no corresponding positive values in the same operand length. The result returned on an overflow is the original src operand.

Integer borrow and overflow conditions are defined in Section 3.1.

Traps: None.

Examples:

1. NEGB R5, R6 4E A0 29
2. NEGW 4(SB), 6(SB) 4E A1 D6 04 06

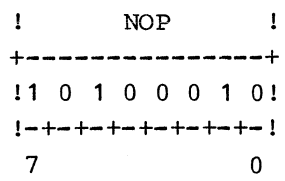
Example 1 negates the low-order byte of register R5 and places the result in the low-order byte of register R6. The remaining bytes of registers R5 and R6 are neither used nor affected.

Example 2 negates the word at the memory address specified by 4(SB) and places the word result at the memory address specified by 6(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R5	AAAAAAFF (-1)	AAAAAAFF (-1)
	R6	BBBBBBBB	BBBBBB01 (+1)
	UPSR	nzfxxltc	nz0xxlt1
Ex. 2:	4(SB)	0041 (+65)	0041 (+65)
	6(SB)	xxxx	FFBF (-65)
	UPSR	nzfxxltc	nz0xxlt1

No Operation

Syntax: **NOP**



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

Flags Affected: None.

Traps: None.

Example:

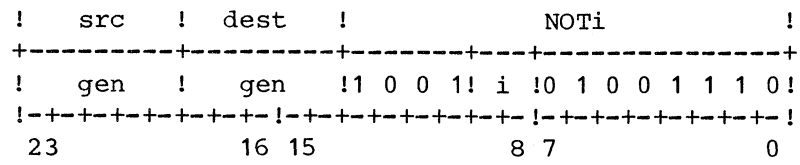
```
NOP                                    A2
```

NOTi

Complement Boolean

Syntax: **NOTi src, dest** NOTB

	gen	gen	NOTW
	read.i	write.i	NOTD



The NOTi instruction complements (inverts) the Boolean value of the src operand and places the result in the dest operand location. The complement of a Boolean value is that value with its least-significant bit complemented. The Boolean value "True" (the integer value 1) thus becomes "False" (the integer value 0) and vice versa. Boolean values are described in Section 3.4.

Flags Affected: None.

Traps: None.

Examples:

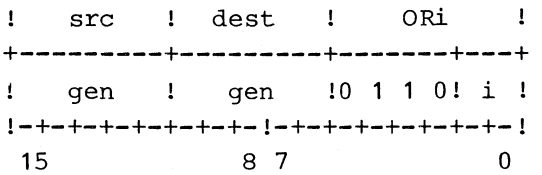
```
1.  NOTB  R0, R0              4E 24 00
2.  NOTW  10(R1), TOS         4E E5 4D 0A
```

Example 1 complements the Boolean value in the low-order byte of register R0. The remaining bytes of R0 are neither used nor affected.

Example 2 complements the 1-word Boolean value at memory address 10(R1) and pushes the result as a word onto the top of the stack.

	Operands	Operand Values: Hex (Boolean) Before	After
Ex. 1:	R0	AAAAAA01 (True)	AAAAAA00 (False)
Ex. 2:	10(R1)	AAAA0000 (False)	AAAA0000 (False)
	Stack:		
	0000FFE0	xxxx	0001 (True)
	0000FFE2	AAAA	AAAA
	SP	0000FFE2	0000FFE0

Syntax: **ORi** **src,** **dest** ORB
gen gen ORW
read.i rmw.i ORD



The ORi instruction performs a bit-wise logical OR operation between the src and dest operands and places the result in the dest operand location.

The instruction ORs each bit in src with the corresponding dest bit. If two corresponding bits are both "0", the dest bit is set to "0"; otherwise, the dest bit is set to "1".

Flags Affected: None.

Traps: None.

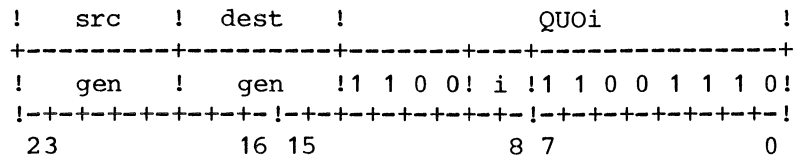
Example:

```
ORB  -6(FP), 11(SB)                98 C6 7A 0B
```

This example ORs the bytes at memory addresses -6(FP) and 11(SB), placing the result at memory address 11(SB).

Operands	Operand Values: Binary Before	After
-6(FP)	11011000	11011000
11(SB)	00001011	11011011

Syntax: **QUOi** **src,** **dest** QUOB
gen gen QUOW
read.i rmw.i QUOD



The QUOi instruction divides the dest operand by the src operand, returning as the result the nearest integer whose absolute value is less than or equal to the absolute value of the exact quotient. The result is placed in the dest operand location.

The src and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: Divide by Zero trap (DVZ) is activated if src equals zero.

Examples:

1. QUOB R0, R7 CE F0 01
2. QUOW 4(SB), 8(SB) CE B1 D6 04 08

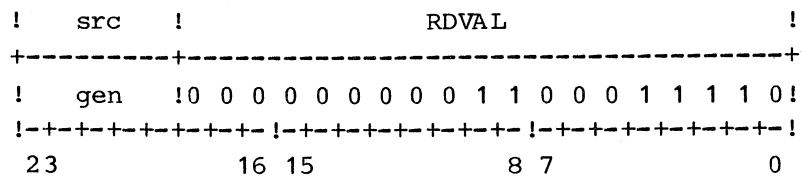
Example 1 divides the low-order byte of register R7 by the low-order byte of register R0, placing the result in the low-order byte of register R7.

Example 2 divides the word at address 8(SB) by the word at address 4(SB), placing the one-word result at address 8(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R0	AAAAAA05 (+5)	AAAAAA05 (+5)
	R7	BBBBBB22 (+34)	BBBBBB06 (+6)
Ex. 2:	4(SB)	0010 (+16)	0010 (+16)
	8(SB)	FFE1 (-31)	FFFF (-1)

Validate Address for Reading

Syntax: **RDVAL** **loc**
gen
addr



The RDVAL instruction checks the protection level assigned to the user-mode virtual memory address specified as loc. If the address is allowed to be read while the CPU is in user mode, the F flag in the PSR is cleared. If the address is not allowed to be read, the F flag in the PSR is set. An address which is protected against reading is also protected against writing, and is therefore inaccessible for any use by a user-mode program.

NOTE: Although the final effective address of loc is interpreted as a user-mode virtual address, any memory references required in order to calculate that effective address are interpreted as using supervisor-mode addresses. This will occur in using the Memory Relative and External addressing modes for loc.

Flags Affected: F is set if loc is inaccessible in user mode, cleared otherwise.

Traps: Undefined Instruction Trap (UND) is activated if the M bit in the CFG register is clear.

Illegal Operation Trap (ILL) is activated if this instruction is attempted while the PSR U bit is set.

Abort Trap (ABT) is activated if the Level 1 page table entry for loc is invalid (V bit = 0). No trap is issued for an invalid Level 2 page table entry, and the Protection Level (PL) field is assumed to be present regardless of the state of the V bit.

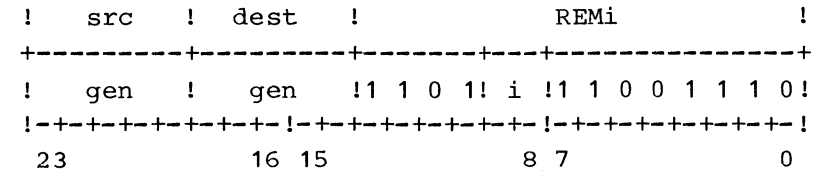
Example:

```
RDVAL 512(R0)                    1E 03 40 82 00
```

This example checks the protection level assigned to the address 512(R0) and sets or clears the F flag to indicate the result.

REMi
Remainder

Syntax: **REMi** **src,** **dest** REMB
gen gen REMW
read.i rmw.i REMD



The REMi instruction places the remainder from dividing dest by src into the dest operand location. The remainder is computed as

dest - src * (dest QUO src) ,

where dest QUO src is the result of dividing dest by src as per the QUOi instruction (q.v.).

The result of a REMi instruction always has the sign of the dest operand (i.e. the dividend) unless the result is zero, which is always positive. Compare the MODi instruction (q.v.).

The src and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: Divide by Zero trap (DVZ) is activated if src equals zero.

Remainder (continued)

Example:

```
REMB  4(SB), 8(SB)                CE B4 D6 04 08
```

This example computes the remainder from dividing the 1-byte operand at address 8(SB) by the 1-byte operand at address 4(SB) and places the result as a byte at address 8(SB).

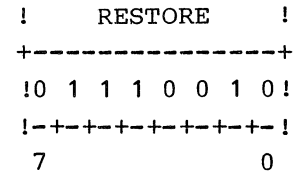
The action of this instruction for four different cases is as follows:

	Operands	Operand Values: Hex (Dec) Before	After
Case 1:	4(SB)	0F (+15)	0F (+15)
	8(SB)	21 (+33)	03 (+3)
Case 2:	4(SB)	F1 (-15)	F1 (-15)
	8(SB)	21 (+33)	03 (+3)
Case 3:	4(SB)	F1 (-15)	F1 (-15)
	8(SB)	DF (-33)	FD (-3)
Case 4:	4(SB)	0F (+15)	0F (+15)
	8(SB)	DF (-33)	FD (-3)

RESTORE

Restore General Purpose Registers

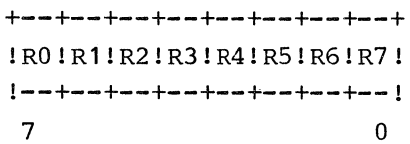
Syntax: **RESTORE** **reglist**
imm



The RESTORE registers instruction restores from the current stack the General Purpose registers specified by reglist.

In assembly language, the reglist operand is specified as a list of zero or more General-Purpose register names enclosed in brackets "[]". The instruction copies to each register in the list a double-word popped from the stack. Register names may appear in any order within reglist but must be separated by commas. Brackets are required even if no register names are given.

In the machine instruction, the reglist operand is encoded in an 8-bit field as shown below. Each bit in the field corresponds to one General-Purpose register. When the instruction is executed, the instruction reads the bits in the field from right to left beginning with bit 0. If a bit is "0", the instruction ignores the corresponding register. If a bit is "1", it restores the corresponding register from the stack. Note that the binary format of the reglist operand is backward from the format of the reglist operand in the SAVE instruction; i.e., bit 0 corresponds to R7 instead of R0.



Flags Affected: None.

Traps: None

Example:

```
RESTORE  [R0, R2, R7]              72 A1
```

This instruction restores the contents of registers R0, R2, and R7 from the stack. The registers are restored in order beginning with register R7 and ending with R0.

Restore General Purpose Registers (continued)

The action of the instruction is illustrated below.

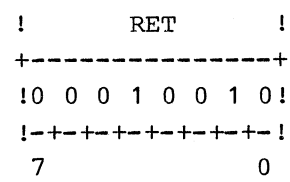
Operands	Operand Values: Hex Before	After
R0	BBBBBBBB	00000010
R2	BBBBBBBB	FFFFFFEF
R7	BBBBBBBB	FFFFF9AB
SP	0000FFE0	0000FFEC
Stack:		
0000FFE0	FFFFF9AB	xxxxxxxx *
0000FFE4	FFFFFFEF	xxxxxxxx *
0000FFE8	00000010	xxxxxxxx *
0000FFEC	AAAAAAAA	AAAAAAAA

* The RESTORE instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

RET

Return from Subroutine

Syntax: **RET** **constant**
disp



The RET instruction returns execution control from a local procedure and removes procedure parameters from the stack.

The instruction pops the return address as a 32-bit value from the currently-selected stack. It then removes the number of bytes specified by the constant operand from the stack by adding the constant operand to the current stack pointer register. Finally, it transfers control by loading the return address into the PC register.

Flags Affected: None.

Traps: None

Return from Subroutine (continued)

Example:

```
RET 16                                  12 10
```

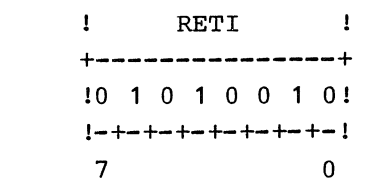
This example pops a new address from the currently-selected stack into the PC and adds 16 (H'10) to the stack pointer.

Operand	Operand Values: Hex Before	After
16 (disp)	10 (+16)	--
PC	00009000	00009010
SP	0000F000	0000F014
Stack:		
0000F000	00009010	xxxxxxxx *
0000F004	BBBBBBBB	xxxxxxxx *
0000F008	BBBBBBBB	xxxxxxxx *
0000F00C	BBBBBBBB	xxxxxxxx *
0000F010	BBBBBBBB	xxxxxxxx *
0000F014	AAAAAAAA	AAAAAAAA

* The RET instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

RETI
Return from Interrupt

Syntax: RETI



The RETI instruction returns control from an interrupt service procedure to the program during which the interrupt was accepted, and informs any interrupt control circuitry present in the system that this is being done. See Chapter 6 for details of interrupt service.

The RETI instruction does the following:

1. Performs either one or two "End of Interrupt" bus cycles in order to inform the appropriate Interrupt Controller(s) that this interrupt service procedure is ending. For details of this aspect of the RETI instruction, see Chapter 6 and the data sheets for the NS32202 Interrupt Control Unit and the appropriate CPU.

2. Pops a 32-bit return address from the currently selected stack into the PC register.

3. Pops a 16-bit MOD address from the currently selected stack into the MOD register.

4. Pops a 16-bit PSR value from the currently selected stack into the PSR.

5. Copies the double-word from the memory address contained in the MOD register into the SB register.

Program execution continues at the new address placed in the PC register.

NOTE: The RETI instructon must not be used to return from the Non-Maskable or Non-Vectored interrupts or from any traps (including the Abort trap). Such use can cause anomalies in prioritization of interrupts by Interrupt Control circuits. For these use instead the Return from Trap instruction (RETT, q.v.).

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

Traps: Illegal Instruction Trap (ILL) is activated if this instruction is attempted while the U flag is set.

Return from Interrupt (continued)

Example:

```
RETI                                52
```

This example returns control from an interrupt service procedure.

The action this instruction is illustrated below. Note that the PSR S flag is assumed to be zero at the beginning of the instruction, thus selecting SP0 as the current Stack Pointer. However, note also that after the instruction is completed the CPU is in User mode, the currently-selected Stack Pointer has become SP1, and interrupts are re-enabled.

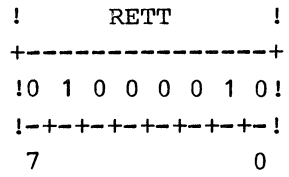
	Operand Values: Hex	
Operands	Before	After
PC	0000F033	00009005
SB	0000F100	00009080
MOD	0020	0010
SP0	00001000	00001008 *
PSR	x000 (xxxxipsu/nzfxxltc)	xB20 (xxxx1011/001xx000)
Stack:		
00001000	00009005	xxxxxxxx **
00001004	0010	xxxx **
00001006	0B20	xxxx **
00001008	AAAA	AAAA
Module Table:		
00000010	00009080 (SB)	00009080 (SB)

* The final Stack Pointer value is the initial address plus 8 as follows: 4 (for double-word return address), 2 (for MOD address), and 2 (for PSR contents).

** The RETI instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

RETT
Return from Trap

Syntax: **RETT** **constant**
disp



The RETT instruction returns control from a trap service procedure. It restores the PC, MOD and PSR registers from the currently-selected stack, updates the SB register, and then removes any parameters passed by the procedure which caused the trap.

The instruction does the following:

1. Pops a 32-bit return address from the currently-selected stack into the PC register.

2. Pops a 16-bit MOD address from the currently-selected stack into the MOD register.

3. Pops a 16-bit PSR value from the currently-selected stack into the PSR. Note that this may switch stack pointers by changing the PSR S bit.

4. Copies the double-word from the address contained in the MOD register into the SB register.

5. Adds the constant operand to the stack pointer newly selected in step 3.

Program execution continues at the new address placed in the PC register.

For a full description of traps, see "Exceptions", Chapter 6.

NOTE: When using the NS32202 Interrupt Control Unit, the RETT instruction must not be used to return from a vectored interrupt, since this instruction does not inform the Interrupt Control Unit that it is returning from an interrupt. To return properly from a vectored interrupt, use the RETI instruction.

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

Traps: Illegal Instruction Trap (ILL) is activated if the U flag is set.

Return from Trap (continued)

Example:

```
RETT  16                              42 10
```

This example returns control from a trap service procedure to a procedure which invoked the trap deliberately after pushing 16 bytes of parameters onto its stack. This instruction removes the 16 bytes from that stack as it returns control.

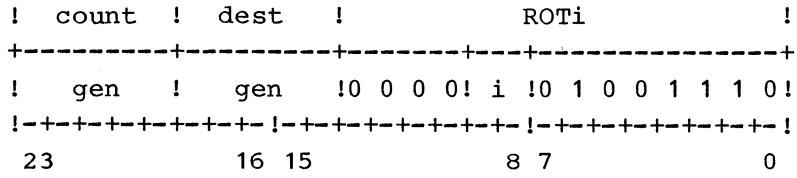
In the following illustration, it is assumed that the trap service routine is using the Interrupt Stack (with SP0 as its stack pointer) and is returning to a procedure which is using the User Stack (with SP1).

Operands	Operand Values: Hex Before	After
16 (disp)	10	--
PC	0000F033	00009005
SB	0000F100	00009080
MOD	0020	0010
SP0	00001018	00001020
SP1	0000FFE0	0000FFF0
PSR	x000 (xxxxipsu/nzfxxltc)	x320 (xxxx0011/001xx000)

Interrupt Stack	00001018	00009005	xxxxxxxx *
	0000101C	03200010	xxxxxxxx *
	00001020	AAAAAAAA	AAAAAAAA
User Stack	0000FFE0	BBBBBBBB	xxxxxxxx *
	0000FFE4	BBBBBBBB	xxxxxxxx *
	0000FFE8	BBBBBBBB	xxxxxxxx *
	0000FFEC	BBBBBBBB	xxxxxxxx *
	0000FFF0	CCCCCCCC	CCCCCCCC
Module Table	00000010	00009080 (SB)	00009080 (SB)
	00000014	00002000 (LB)	00002000 (LB)
	00000018	00009000 (PB)	00009000 (PB)

* The RETT instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

Syntax: **ROTi** **count,** **dest** ROTB
gen gen ROTW
read.B rmw.i ROTD



The ROTi instruction performs a rotation shift on the dest operand in the manner specified by the count operand. The sign of count determines the direction of the shift. The absolute value of count gives the number of bit positions to shift the dest operand.

The count operand value must be within the range -7 to +7 for the ROTB form, -15 to +15 for the ROTW form, and -31 to +31 for the ROTD form. A positive count specifies a left shift; a negative count specifies a right shift. In a rotation, each bit rotated off one end of dest is moved to the emptied bit position at the other end of dest.

The count operand is interpreted as a signed integer. The dest operand is interpreted as an unsigned integer.

Flags Affected: None.

Traps: None.

Rotate (continued)

Examples:

```
1.  ROTB   4, R5                4E 40 A1 04
2.  ROTB  -3, 16(SP)            4E 40 A6 FD 10
```

Example 1 rotates the least-significant byte of register R5 four bit positions to the left. The remaining bytes of R5 are unaffected.

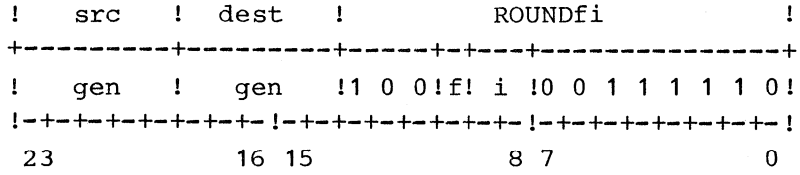
Example 2 rotates the operand at address 16(SP) three bit positions to the right.

	Operands	Operand Values: Before	Binary (Dec) After
Ex. 1:	4 (immediate)	00000100 (+4)	--------
	R5 (low byte)	00001111	11110000
Ex. 2:	-3 (immediate)	11111101 (-3)	--------
	16(SP)	00001111	11100001

ROUNDfi
Round Floating to Integer

Syntax: **ROUNDfi** **src,** **dest** ROUNDFB ROUNDLB

	gen	gen	ROUNDFW ROUNDLW
	read.f	write.i	ROUNDFD ROUNDLD



The Round Floating to Integer instruction rounds the src operand to the nearest integer and places the result in the dest operand location as a signed integer. If src is exactly halfway between two integer values, it is rounded to the even value (i.e. the value that is exactly divisible by two).

Flags Affected: No PSR flags. FSR flags are affected as follows:
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant to this instruction is the Overflow exception, which is caused by attempting to convert a floating-point number that is too great in absolute value to be held in a signed integer of the size specified for dest.

Round Floating to Integer (continued)

Examples:

```
1.  ROUNDFB  F0, R0              3E 24 00
2.  ROUNDLD  F2, 12(SB)          3E A3 16 0C
```

Example 1 rounds the single-precision number in register F0 to a 1-byte integer and places the result in the low-order byte of register R0. The remaining bytes of R0 are unaffected.

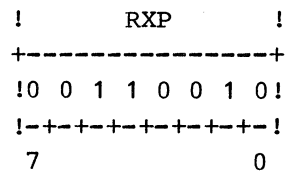
Example 2 rounds the double-precision number in register pair (F2,F3) to a double-word integer and places the result at address 12(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	40180000 (+2.375)	40180000 (+2.375)
	R0	AAAAAAAA	AAAAAA02 (+2)
Ex. 2:	(F2,F3)	41C0200888700000 (+541069584.875)	41C0200888700000 (+541069584.875)
	12(SB)	AAAAAAAA	20401111 (+541069585)

RXP

Return from External Procedure

Syntax: **RXP** **constant**
disp



The RXP instruction returns control from an externally-called procedure and removes any procedure parameters from the stack.

The instruction does the following:

1. Pops a 32-bit return address from the currently-selected stack into the PC register.

2. Pops a 16-bit MOD address from the currently-selected stack to the MOD register and increments the stack pointer by two. The stack pointer is modified by a total of four in this step.

3. Copies the double-word at address MOD+0 to the SB register.

4. Adds the constant operand to the current stack pointer.

Program execution continues at the new address placed in the PC register.

Flags Affected: None.

Traps: None.

Return from External Procedure (continued)

Example:

```
RXP  16                          32 10
```

This example returns control from an externally-called procedure and removes 16 (H'10) bytes from the currently-selected stack.

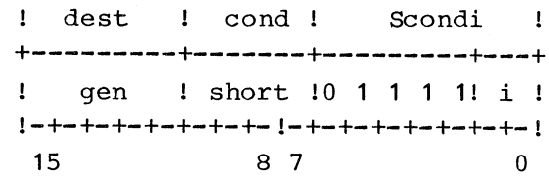
Operands	Operand Values: Hex Before	After
16 (disp)	10 (+16)	--
PC	0000F033	00009005
SB	0000F100	00009080
MOD	0020	0010
SP	00001018	00001030 *
Stack:		
00001018	00009005	xxxxxxxx **
0000101C	xxxx0010	xxxxxxxx **
00001020	BBBBBBBB	xxxxxxxx **
00001024	BBBBBBBB	xxxxxxxx **
00001028	BBBBBBBB	xxxxxxxx **
0000102C	BBBBBBBB	xxxxxxxx **
00001030	AAAAAAAA	AAAAAAAA
Module Table:		
00000010	00009080 (SB)	00009080 (SB)
00000014	00002000 (LB)	00002000 (LB)
00000018	00009000 (PB)	00009000 (PB)

* The final Stack Pointer content is the initial address plus 4 (for double-word return address), 2 (for word MOD address), 2 (additional from step 3), and 16 as specified by the constant operand.

** The RXP instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

Scondi
Save Condition as Boolean

Syntax: **Scondi** **dest** ScondB
gen ScondW
write.i ScondD



The Scondi instruction sets the dest operand to the integer value "1" if the specified condition is true, and to "0" if false. These are the Boolean values "True" and "False", respectively.

Cond is a 2-character condition name that specifies the state of a flag or flags in the PSR. If the flag(s) have the specified state, the condition is true; otherwise, the condition is false.

The Save Condition as Boolean instruction may specify the following conditions:

Condition	Condition Name	True State	Short Field
Equal	EQ	Z flag set	0000
Not Equal	NE	Z flag clear	0001
Carry Set	CS	C flag set	0010
Carry Clear	CC	C flag clear	0011
Higher	HI	L flag set	0100
Lower or Same	LS	L flag clear	0101
Greater Than	GT	N flag set	0110
Less Than or Equal	LE	N flag clear	0111
Flag Set	FS	F flag set	1000
Flag Clear	FC	F flag clear	1001
Lower	LO	Z and L flag clear	1010
Higher or Same	HS	Z or L flag set	1011
Less Than	LT	Z and N flag clear	1100
Greater than or Equal	GE	Z or N flag set	1101

The condition name must be embedded in the instruction mnemonic as illustrated in the examples below. The name is translated at assembly time to the corresponding Short Field of the basic instruction.

The interpretation of condition codes is such that the instruction sequence

```
CMPB   A,B
SGTB   RESULT
```

will store "1" in RESULT if operand A is greater than operand B in the CMPB instruction.

Save Condition as Boolean (continued)

Flags Affected: None.

Traps: None.

Examples:

```
1.  SEQB  R0          3C 00
2.  SLOW  10(SB)      3D D5 0A
3.  SHID  TOS         3F BA
```

Example 1 sets the low-order byte of register R0 to 1 if the Z flag is set, 0 if the Z flag is clear.

Example 2 sets the word at the operand address 10(SB) to 1 if the Z and L flags are clear, 0 otherwise.

Example 3 pushes a double-word value onto the stack: 1 if the L flag is set, 0 otherwise.

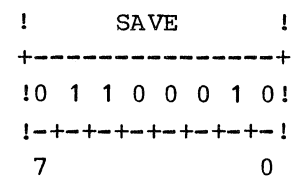
In the following illustration, the Z and L flags are assumed to be set.

	Operands	Operand Values: Hex (Boolean) Before	After
Ex. 1:	R0	AAAAAAAA	AAAAAA01 (True)
	UPSR	n1fxx1tc	n1fxx1tc
Ex. 2:	10(SB)	AAAA	0000 (False)
	UPSR	n1fxx1tc	n1fxx1tc
Ex. 3:	Stack:		
	00001000	xxxxxxxx	00000001 (True)
	00001004	AAAAAAAA	AAAAAAAA
	SP	00001004	00001000
	UPSR	n1fxx1tc	n1fxx1tc

SAVE

Save General Purpose Registers

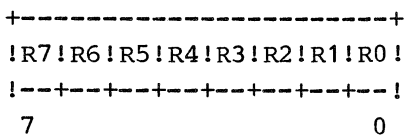
Syntax: **SAVE** **reglist**
imm



The SAVE instruction saves the General-Purpose registers specified by reglist, pushing them onto the currently-selected stack.

The reglist operand is a list of zero or more general purpose register names, enclosed in brackets "[]". The instruction pushes the contents of each register in the list as a double-word onto the stack. Register names may appear in any order within reglist, but must be separated by commas. Brackets are required even if no register names are given.

In the machine instruction, the reglist operand is encoded in an 8-bit field as shown below. Each bit in the field corresponds to one general purpose register. When the instruction is executed, the instruction reads the bits in the field from right to left beginning with bit 0. If a bit is "0", the instruction ignores the corresponding register. If a bit is "1", it pushes the corresponding register.



Flags Affected: None.

Traps: None.

Example:

```
SAVE   [R0, R2, R7]                62 85
```

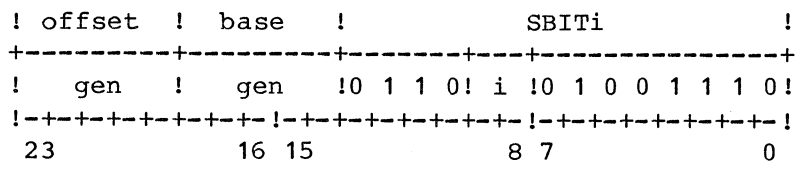
This instruction saves the contents of registers R0, R2, and R7 on the stack. The registers are stored in order beginning with register R0 and ending with R7.

Save General Purpose Registers (continued)

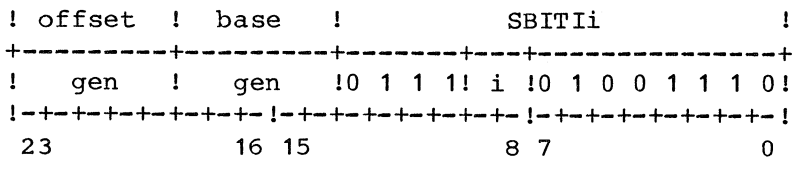
Operands	Operand Values: Hex Before	After
R0	00000010	00000010
R2	FFFFFFEF	FFFFFFEF
R7	FFFFF9AB	FFFFF9AB
SP	0000FFEC	0000FFE0
Stack:		
0000FFE0	xxxxxxxx	FFFFF9AB
0000FFE4	xxxxxxxx	FFFFFFEF
0000FFE8	xxxxxxxx	00000010
0000FFEC	AAAAAAAA	AAAAAAAA

SBITi
SBITIi
Set Bit, Set Bit Interlocked

Syntax: **SBITi** **offset, base** SBITB SBITIB
gen gen SBITW SBITIW
read.i regaddr SBITD SBITID



Syntax: **SBITIi** **offset, base**
gen gen
read.i regaddr



The SBITi and SBITIi instructions set to 1 the register or memory bit specified by base and offset after copying the bit value to the F flag in the PSR.

The SBITIB, SBITIW, and SBITID instructions, in addition, activate the Interlocked Operation output pin on the CPU, which may be used in multiprocessor systems to interlock accesses to semaphore bits. See the applicable CPU data sheet for further details.

The location of the bit is determined from offset and base. Offset is a general operand, whose length is given by the operation length suffix. Base is an addressing expression giving a byte address from which offset specifies a bit position. See Section 3.5 for details of specifying bit positions.

If base is a register, then the bit is within that register, at the bit position given by the offset operand. If base is a memory location, then the bit is at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.5 for definitions of the operators MOD and DIV above, and for further details of bit instructions.

Offset is interpreted as a signed integer.

Set Bit (continued)

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

Traps: None.

Example:

```
SBITW  R0, 1(R1)                    4E 59 02 01
```

This example sets a bit in memory to 1 after copying the bit value to the F flag. This performs the basic operation of a semaphore "Test and Set". In a multiprocessor system, the SBITIW instruction would have been used instead. For designating the location of the target bit, the low-order word of register R0 supplies the bit offset, and 1(R1) is specified as the base address.

In the following illustration, the target bit is assumed to be 0 prior to instruction execution.

Operands	Operand Values: Hex (Dec) [Binary] Before	After
R0 (offset)	AAAA004C (+76)	AAAA004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 1(R1)	00001001 (+4097)	--
0000100A * (+4106)	EF [11101111]	FF [11111111]
UPSR	nzfxxltc	nz0xxltc

* The address 100A (Hex) is the effective address of the byte containing the desired bit. This address is computed from the offset and the base address as follows:

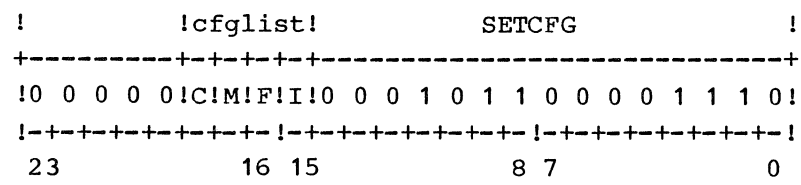
```
base address + (offset DIV 8)
   4097      +       9
   4106, or 100A (Hex)  .
```

The bit number within this byte is calculated as:

```
offset MOD 8
  76   MOD 8
       4  .
```

SETCFG
Set Configuration

Syntax: **SETCFG** **cfglist**
short



The SETCFG instruction loads the Configuration Register (CFG), enabling or disabling optional system features.

In assembly language, cfglist is a list of zero or more configuration bit names. The names are I, F, M and C. The names may appear in any order in the list, but must be separated by commas. The list itself must be enclosed in brackets. Brackets are required even if no bit name is given.

The cfglist operand is held in a 4-bit field in the basic instruction, as shown above. Each bit corresponds to one bit in the CFG register.

If I is specified, the I bit in the CFG register is set and vectored interrupt processing is enabled. (An NS32202 Interrupt Control Unit or equivalent must be present in the system.) Otherwise, the I bit is cleared and all maskable interrupts are serviced as non-vectored interrupts. See Chapter 6 for interrupt handling modes.

If F is specified, the F bit in the CFG register is set and Floating Point instructions are available. (An NS32081 Floating-Point Unit must be present in the system.) Otherwise, the F bit is cleared and Floating Point instructions activate the Undefined Instruction Trap (UND).

If M is specified, the M bit in the CFG register is set and Memory Management instructions are available. (An NS32082 Memory Management Unit must be present in the system.) Otherwise, the M bit is cleared and Memory Management instructions activate the Undefined Instruction Trap (UND).

If C is specified, the C bit in the CFG register is set and Custom Slave instructions are available. (System-dependent Custom Slave hardware must be present to execute the instructions.) Otherwise, the C bit is cleared and Custom Slave instructions activate the Undefined Instruction Trap (UND).

Set Configuration (continued)

NOTES:
1. A CFG bit name may only be specified if the corresponding device is present in the system.
2. The state of the M bit in the CFG register does not directly affect address translation hardware or bus timing. It only enables or disables the Memory Management instruction set.
3. When a Floating-Point, Memory Management, or Custom Slave instruction activates an Undefined Instruction Trap (UND) (i.e, when the corresponding CFG register bit is 0), it is possible to intercept the trap and simulate the instruction in software.

Flags Affected: None.

Traps: Illegal Instruction Trap (ILL) is activated if this instruction is attempted while the U flag is set.

Example:

```
SETCFG  [I,M,F]                    0E 8B 03
```

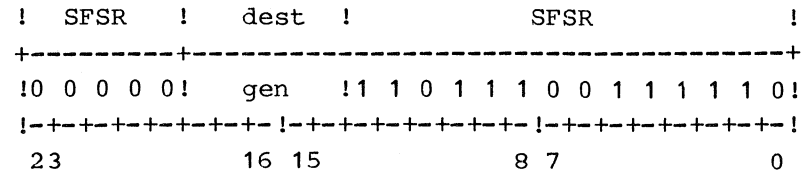
This instruction sets the I, M, and F bits in the CFG register, enabling vectored interrupt processing and the Memory Management and Floating Point instruction sets. The C bit is cleared, disabling the Custom Slave instruction set.

	Bit Values	
Operands	Before	After
CFG	cmfi	0111

SFSR

Store Floating-Point Status Register (FSR)

Syntax: **SFSR** **dest**
gen
write.D



The SFSR instruction copies the contents of the Floating-Point Status Register (FSR) to the dest operand location. The FSR is treated as a 32-bit value. See Section 2.4.2 for the format of the FSR.

Flags Affected: None.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Example:

```
SFSR  TOS                          3E F7 05
```

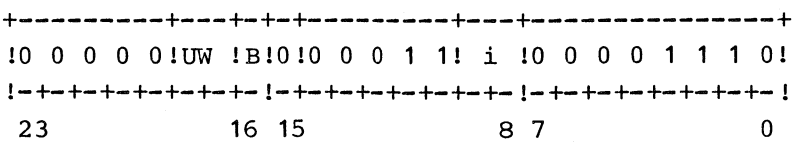
This example pushes the contents of the FSR onto the top of the currently selected stack as a double word.

Operands	Operand Values: Hex Before	After
FSR	xxxx0028	xxxx0028
SP	0000FFDE	0000FFDA
Stack:		
0000FFDA	xxxxxxxx	xxxx0028
0000FFDE	AAAAAAAA	AAAAAAAA

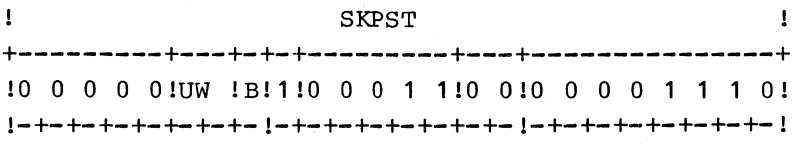
Skip String

Syntax: **SKPSi options**

SKPSB
SKPSW
SKPSD
SKPST



Syntax: **SKPST options**



Operands of the SKPSi and SKPST instructions are specified in General-Purpose registers:

R0 - Number of string elements to be processed.
R1 - Address of current String 1 element.
R2 - (not used)
R3 - Address of translation table (SKPST form only).
R4 - Match value (with Until Match or While Match option only).

The SKPSi instruction examines and skips over consecutive elements in String 1 until either an Until/While condition is met or register R0 is decremented to 0 (i.e., the string is exhausted). After each element is examined, the CPU sets register R1 to the address of the next element to be examined and register R0 to the number of integers remaining to be examined. Register R2 is not used or affected.

The SKPST instruction causes the CPU to internally replace the current String 1 element value with a corresponding translated value before performing its examination. The translated value to be examined is found by adding the current element from String 1 as an unsigned integer to the translation table address found in register R3. The instruction examines elements and sets registers as described above. The SKPST instruction operates on byte-long elements only.

Options may be specified by listing the letters B (Backward), U (Until Match) and W (While Match) as operands. The U and W options are mutually exclusive. See Section 3.7 for details of the options available in String instructions.

In the machine instruction, the options are encoded in the B and UW fields as follows:

B field =	0	Forward direction.
	1	Backward direction.
UW field =	00	Neither Until Match nor While Match.
	01	While Match.
	10	(reserved)
	11	Until Match.

String instructions are interruptible. See Section 3.7.

Flags Affected: F is set if the U or W option is specified and the corresponding Until/While condition is met, otherwise it is cleared.

Traps: None.

Example:

```
SKPSB   U                              0E 0C 06
```

This example examines and skips over byte-long elements in String 1 until the current integer and the contents of the low-order byte of register R4 are equal or until register R0 contains zero.

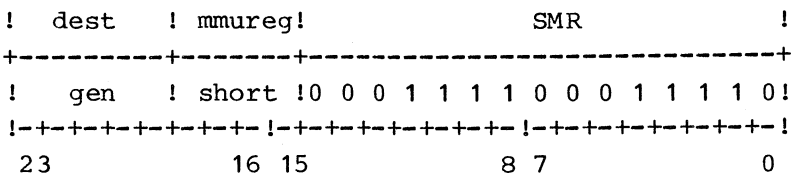
In the following illustration, the underlined string element shows the point at which the instruction terminates.

Operands	Operand Values: Hex (Dec) Before	After
R0	00000020 (+32)	00000016 (+22)
R1	00002000	0000200A
R4	AAAAAA1F (+31)	AAAAAA1F (+31)
UPSR	nzfxxltc	nz1xxltc

Starting Address	String Contents
2000	1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
	1F 1D 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18

Store Memory Management Register

Syntax: **SMR** **mmureg, dest**
short gen
write.D



The SMR instruction copies the contents of the Memory Management register specified by mmureg to the dest operand location.

The Store MMU Register instruction may store the following registers. The short field of the basic instruction holds a 4-bit value which relates to the corresponding mmureg specifications as follows:

Register	mmureg	short field
Breakpoint Register 0	BPR0	0000
Breakpoint Register 1	BPR1	0001
Program Flow Register 0	PF0	0100
Program Flow Register 1	PF1	0101
Sequential Count Registers	SC	1000
Memory Management Status Register	MSR	1010
Breakpoint Count Register	BCNT	1011
Page Table Base Register 0	PTB0	1100
Page Table Base Register 1	PTB1	1101
Error/Invalidate Address Register	EIA	1111

Flags Affected: None.

Traps: Undefined Instruction Trap (UND) is activated if the M bit in the CFG register is clear. The instruction is not executed.

Illegal Instruction Trap (ILL) is activated if the U flag is set. The instruction is not executed.

Example:

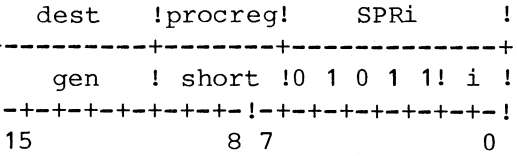
```
SMR  BCNT, R0                    1E 8F 05
```

This example copies the contents of the Breakpoint Count Register in the MMU to register R0.

Operands	Operand Values: Hex Before	After
BCNT	xx009000	xx009000
R0	AAAAAAAA	xx009000

Store Processor Register

Syntax: **SPRi** **procreg, dest** SPRB
short gen SPRW
write.i SPRD



The SPRi instruction stores the contents of the dedicated processor register specified by procreg in the dest operand location.

The register contents are right-justified in dest. High-order dest bits are zero-filled if the register is shorter than dest. High-order register bits are discarded if the register is longer than dest. See Section 2.2 for the formats of the dedicated registers.

The Store Processor Register instruction may store the following registers. The specified procreg corresponds to the 4-bit short field in the basic instruction as shown below.

Register	procreg	short field	
User PSR	UPSR	0000	(Note 2)
Frame Pointer	FP	1000	
Stack Pointer	SP	1001	(Note 1)
Static Base Register	SB	1010	
Program Status Register	PSR	1101	(Note 2)
Interrupt Base Register	INTBASE	1110	(Note 2)
Module Register	MOD	1111	

NOTES:

1. If SP is specified in the instruction and the S flag in the PSR is set, the instruction copies the SP1 register to the dest operand location. If the S flag is clear, the instruction copies the SP0 register to the dest operand location.

2. Specifying PSR or INTBASE as the procreg operand is privileged, regardless of the operation length specified. If UPSR is specified, however, only the low-order eight bits of the PSR are stored, regardless of the operation length specified. They are zero-extended as necessary to fill the dest operand. This is not a privileged operation.

Flags Affected: None.

Traps: Illegal Instruction Trap (ILL) is activated if the U flag is set and PSR or INTBASE is specified.

Examples:

```
1.  SPRD  FP, R0              2F 04
2.  SPRW  MOD, 4(SB)          AD D7 04
```

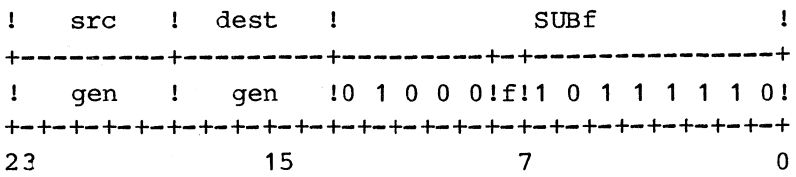
Example 1 copies the entire FP register to register R0.

Example 2 copies the contents of the MOD register to a word at the address specified by 4(SB).

	Operands	Operand Values: Hex Before	After
Ex. 1:	FP	00000021	00000021
	R0	AAAAAAAA	00000021
Ex. 2:	MOD	0030	0030
	4(SB)	AAAA	0030

Subtract Floating

Syntax: **SUBf** **src,** **dest** SUBF
gen gen SUBL
read.f rmw.f



The SUBf instruction subtracts the src operand from the dest operand and places the result in the dest operand location. Subtraction can be modelled as negating the src operand and adding the result to the dest operand. For details of the addition step see the ADDf instruction.

Flags Affected: No PSR flags. FSR flags are affected as follows:
UF is set if an underflow occurs; unaffected otherwise.
IF is set on an inexact result; unaffected otherwise.
TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.
See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3.

Examples:

1.	SUBF F0, F7	BE D1 01
2.	SUBL F2, 16(SB)	BE 90 16 10

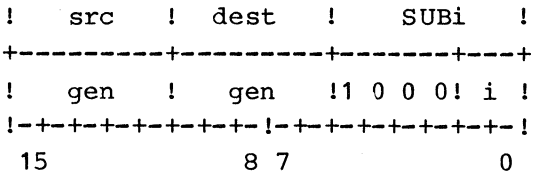
Example 1 subtracts the single-precision number in register F0 from the single-precision number in register F7 and places the result in register F7.

Example 2 subtracts the double-precision number in register pair (F2,F3) from the double-precision number at the address 16(SB) and places the double-precision result at the address 16(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	40840000 (+4.125)	40840000 (+4.125)
	F7	41F50000 (+30.625)	41D40000 (+26.500)
Ex. 2:	(F2,F3)	4114C86300000000 (+340504.750)	4114C86300000000 (+340504.750)
	16(SB)	41C022A194900000 (+541410089.125)	41C0200888300000 (+541069584.375)

Subtract

Syntax: **SUBi** **src,** **dest** SUBB
gen gen SUBW
read.i rmw.i SUBD



The SUBi instruction subtracts the src operand from the dest operand and places the result in the dest operand location.

Flags Affected: C is set on a borrow from subtraction, cleared if no borrow. F is set on an overflow from subtraction, cleared if no overflow.

Integer carry and overflow conditions are defined in Section 3.1.

Traps: None.

Examples:

```
1.  SUBB  R0, R1                    60 00
2.  SUBD  4(SB), 20(SB)             A3 D6 04 14
```

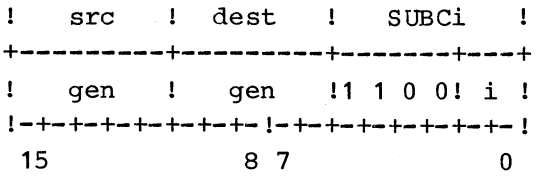
Example 1 subtracts the low-order byte of register R0 from the low-order byte of register R1 and places the result in the low-order byte of register R1. The remaining bytes of R1 are not affected.

Example 2 subtracts the double-word at the memory address specified by 4(SB) from the double-word at the memory address specified by 20(SB). The instruction places the result at memory address 20(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R0	AAAAAA01 (+1)	AAAAAA01 (+1)
	R1	BBBBBB7F (+127)	BBBBBB7E (+126)
	UPSR	nzfxxltc	nz0xxlt1
Ex. 2:	4(SB)	FFFFFFFE (-2)	FFFFFFFE (-2)
	20(SB)	00010000 (+65536)	00010002 (+65538)
	UPSR	nzfxxltc	nz0xxlt1

Subtract with Carry [Borrow]

Syntax: **SUBCi** **src,** **dest** SUBCB
gen gen SUBCW
read.i rmw.i SUBCD



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

Flags Affected: C is set on a borrow from subtraction, cleared if no borrow. F is set on an overflow from subtraction, cleared if no overflow.

Integer carry and overflow conditions are defined in Section 3.1.

Traps: None.

Examples:

```
1.  SUBCB  32, R1                70 A0 20
2.  SUBCW  TOS, -8(FP)           31 BE 78
```

Example 1 subtracts the sum of 32 and the C flag value from the low-order byte of register R1 and places the result in the low-order byte of register R1. The remaining bytes of R1 are not affected.

Example 2 subtracts the sum of the word at the top of the stack and the C flag value from the word at the memory address specified by -8(FP). The instruction then places the 2-byte result at the memory address specified as -8(FP).

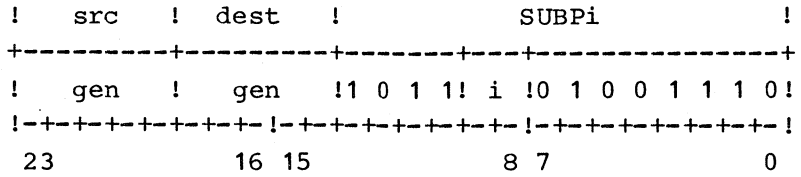
In the following illustration, the C flag value is assumed to be 1.

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	32 (immediate)	20 (+32)	--
	R1	00000050 (+80)	0000002F (+47)
	UPSR	nzfxxlt1	nz0xxlt0
Ex. 2:	-8(FP)	CB99 (-13415)	9286 (-28026)
	UPSR	nzfxxlt1	nz0xxlt0
	Stack:		
	0000FFEE	3912 (+14610)	xxxx *
	0000FFF0	AAAA	AAAA
	SP	0000FFEE	0000FFF0

* The instruction has not itself changed the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

Subtract Packed Decimal

Syntax: **SUBPi** **src,** **dest** SUBPB
gen gen SUBPW
read.i rmw.i SUBPD



The SUBPi instruction subtracts the src operand from the dest operand and then subtracts the C flag. The instruction places the result in the dest operand location as a packed decimal (BCD) integer.

The src and dest operands are interpreted as unsigned packed decimal (BCD) integers. If either operand contains invalid digits, the result is undefined. See Section 3.2 for details of packed decimal arithmetic.

Flags Affected: C is set on a borrow from subtraction, cleared if no borrow.
F is cleared.

Traps: None.

Examples:

```
1.  SUBPD  H'99, R1              4E 6F A0 00 00 00 99
2.  SUBPB  -8(FP), 16(FP)        4E 2C C6 78 10
```

Example 1 subtracts the packed decimal integer 99 from the contents of register R1 and then subtracts the C flag. The result is placed in resister R1.

Example 2 subtracts the packed decimal integer at memory address -8(FP) from the packed decimal integer at memory address 16(FP) and then subtracts the C Flag. The instruction places the result at memory address 16(FP).

In the following illustration, the C flag value is assumed to be 0.

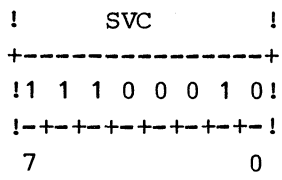
	Operands	Operand Values: Hex * Before	After
Ex. 1:	H'99 (immediate)	00 00 00 99	--
	R1	00000187	00000088
	UPSR	nzfxxlt0	nz0xxlt0
Ex. 2:	-8(FP)	10	10
	16(FP)	01	91 **
	UPSR	nzfxxlt0	nz0xxlt1

* The hexadecimal representation also expresses the decimal interpretation of the value.

** In Example 2, subtraction results in a borrow.

Supervisor Call

Syntax: **SVC**



The SVC instruction activates the Supervisor Call Trap (SVC). The Supervisor Call Trap passes program execution control to the SVC service procedure (see "Exceptions", Chapter 6.) The return address pushed onto the Interrupt Stack is the address of the SVC instruction itself.

Flags Affected: None.

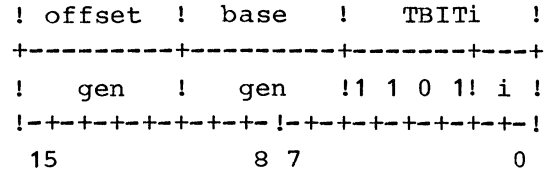
Traps: Supervisor Call Trap (SVC) is activated.

Example:

```
SVC                                 E2
```

TBITi
Test Bit

Syntax: **TBITi** **offset,** **base** TBITB
gen gen TBITW
read.i regaddr TBITD



The TBITi instruction copies the register or memory bit specified by base and offset to the F flag.

The location of the bit is determined from offset and base. Offset is a general operand, whose length is given by the operation length suffix. Base is an addressing expression giving a byte address from which offset specifies a bit position. See Section 3.5 for details of specifying bit positions.

If base is a register, then the bit is within that register, at the bit position given by the offset operand. If base is a memory location, then the bit is at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.5 for definitions of the operators MOD and DIV above, and for further details of bit instructions.

Offset is interpreted as a signed integer.

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

Traps: None.

Test Bit (continued)

Example:

```
TBITW  R0, 0(R1)                    75 02 00
```

This example copies a bit from memory to the F flag. The low-order word of register R0 supplies the bit offset, and 0(R1) is specified as the base address.

In the following illustration, the target bit is assumed to be 1.

Operands	Operand Values: Hex (Dec) [Binary] Before	After
R0 (offset)	AAAA004C (+76)	AAAA004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 0(R1)	00001000 (+4096)	--
00001009 * (+4105)	10 [00010000]	10 [00010000]
UPSR	nzfxxltc	nz1xxltc

* The address 1009 (Hex) is the effective address of the byte containing the desired bit. This address is computed from the offset and the base address as follows:

```
base address + (offset DIV 8)
   4096      +       9
   4105, or 1009 (Hex)  .
```

The bit number within this byte is calculated as:

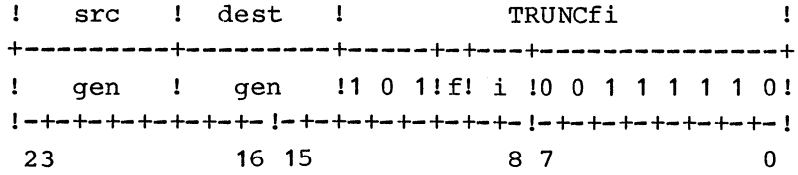
```
offset MOD 8
  76   MOD 8
       4  .
```

TRUNCfi

Truncate Floating to Integer

Syntax: **TRUNCfi** **src,** **dest**

	gen	gen	TRUNCFB TRUNCLB
	read.f	write.i	TRUNCFW TRUNCLW
			TRUNCFD TRUNCLD



The TRUNCfi instruction truncates the src operand to the nearest integer which is less than or equal to it in absolute value and places the result in the dest operand location as a signed integer.

Flags Affected: No PSR flags. FSR flags are affected as follows:

IF is set on an inexact result; unaffected otherwise.

TT field is set to reflect any exceptional conditions encountered in executing the instruction. If none is encountered, TT is set to all zeroes.

See Sections 2.4.2 and 3.3 for details of exceptional conditions and reporting.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant to this instruction is the Overflow exception, which is caused by attempting to convert a floating-point number which is too great in absolute value to be held in a signed integer of the size specified for dest.

Truncate Floating to Integer (continued)

Examples:

```
1.  TRUNCFB  F0, R0              3E 2C 00
2.  TRUNCLD  F2, 8(SB)           3E AB 16 08
```

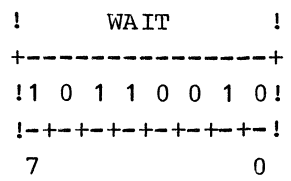
Example 1 truncates the single-precision number in register F0 to a one-byte integer and copies the integer to the low-order byte of register R0.

Example 2 truncates the double-precision number in register pair (F2,F3) to a double-word integer and copies the integer to address 8(SB).

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	C0280000 (-2.625)	C0280000 (-2.625)
	R0	AAAAAAAA	AAAAAAFE (-2)
Ex. 2:	(F2,F3)	41C0200888700000 (+541069584.875)	41C0200888700000 (+541069584.875)
	8(SB)	AAAAAAAA	20401110 (+541069584)

WAIT
Wait

Syntax: **WAIT**



The WAIT instruction suspends program execution until an interrupt occurs. An interrupt restores program execution by passing it to an interrupt service procedure. Interrupts are described in "Exceptions", Chapter 6. When the WAIT instruction is interrupted, the return address saved is the address of the instruction following the WAIT instruction.

Flag Affected: None.

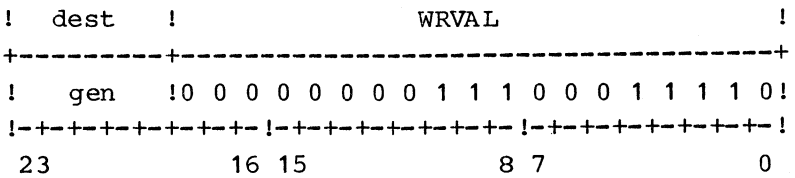
Traps: None.

Example:

```
WAIT                                B2
```

Validate Address for Writing

Syntax: **WRVAL** **loc**
gen
addr



The WRVAL instruction checks the protection level assigned to the user-mode virtual memory address specified as loc. If the address can be written to while in user mode, the F flag in the PSR is cleared. If the address cannot be written to (i.e., if loc is write-protected), the F flag in the PSR is set. See Section 3.12 for details of Memory Management instructions.

NOTE: Although the final effective address of loc is interpreted as a user-mode virtual address, any memory references required in order to calculate that effective address are interpreted as using supervisor-mode addresses. This will occur in using the Memory Relative and External addressing modes for loc.

Flags Affected: F is set if loc is write-protected, cleared otherwise.

Traps: Undefined Instruction Trap (UND) is activated if the M bit in the CFG register is clear.

Illegal Operation Trap (ILL) is activated if this instruction is attempted while the PSR U flag is set.

Abort Trap (ABT) is activated if the Level 1 page table entry for loc is invalid (V bit = 0). No trap is issued for an invalid Level 2 page table entry, and the Protection Level (PL) field is assumed to be present regardless of the state of the V bit.

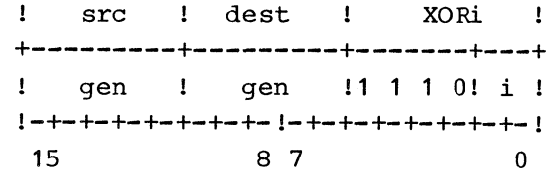
Example:

```
WRVAL 512(R0)                    1E 07 40 82 00
```

This example checks the protection level assigned to the user-mode virtual address 512(R0) and sets or clears the F flag to indicate the result.

XORi
Exclusive Or

Syntax: **XORi** **src,** **dest** XORB
gen gen XORW
read.i rmw.i XORD



The XORi instruction performs a bit-wise Exclusive-OR operation on the src and dest operands and places the result in the dest operand location.

The instruction XORs each bit of src with the corresponding dest bit. If two corresponding bits are equal, the dest bit is set to "0"; otherwise, the dest bit is set to "1".

Flags Affected: None.

Traps: None.

Example:

```
XORB  -8(FP), -4(FP)              38 C6 78 7C
```

This example XORs the bytes at the addresses specified by -8(FP) and -4(FP) and places the result in the byte at address -4(FP).

Operands	Operand Values: Binary Before	After
-8(FP)	11110000	11110000
-4(FP)	10010101	01100101

Chapter 6

EXCEPTION PROCESSING

Exceptions processed in a Series 32000 system are of three general types.

A Reset is performed whenever an RST signal is received by the processor. It immediately terminates all other processing and reinitializes the processor state.

Interrupt service is performed by a Series 32000 CPU upon receipt of an interrupt request on either of two input pins:

INT, on which maskable interrupts are requested, and

NMI, on which non-maskable interrupts are requested.

Traps comprise a group of special interrupts which are triggered as a result either of exceptional conditions encountered in executing an instruction (e.g., attempted division by zero) or of specific instructions whose purpose is to cause a trap to occur (e.g., the Supervisor Call instruction).

6.1 Reset

Resetting a Series 32000 system has the following effects:

1. All implemented bits of the PC, PSR and FSR registers are cleared.

2. In the MSR, the NT, FT, BEN, TS, TU and ERC fields are cleared.

All other register contents are undefined. Program execution begins at address zero. (In memory-managed systems, this is physical address zero.)

6.2 General Interrupt/Trap Sequence

Upon receipt of an interrupt or trap request, the CPU goes through four major steps:

1. Adjustment of Registers. Depending on the source of the interrupt or trap, the CPU may update and/or adjust the contents of the Program Counter (PC), the Processor Status Register (PSR) and the currently-selected Stack Pointer (SP). A copy of the PSR is made, and the PSR is then set to reflect Supervisor Mode and selection of the Interrupt Stack.

2. Saving Processor Status. The PSR copy is pushed onto the Interrupt Stack as a 16-bit value.

3. Vector Acquisition. A vector number is either read from a source external to the CPU or is supplied internally.

4. Service Call. The vector number is used as an index into the Interrupt Dispatch Table, whose base address is taken from the CPU Interrupt Base (INTBASE) Register. See Figure 6-1. A 32-bit external procedure descriptor is read from this table entry, and an external procedure call is performed using it. The MOD Register (16 bits) and Program Counter (32 bits) are pushed onto the Interrupt Stack as part of this step.

This process is illustrated in Figure 6-2.

Full sequences of events in processing interrupts and traps may be found as follows:

Maskable and Non-Maskable Interrupts:	Section 6.8.1
Abort Trap:	Section 6.8.4
Trace Trap:	Section 6.8.3
Other Traps:	Section 6.8.2

6.3 Interrupt/Trap Return

To return control to an interrupted program, one of two instructions is used. The RETT (Return from Trap) instruction restores the PSR, MOD, PC and SB registers to their previous contents and, since traps are often used deliberately as a call mechanism for Supervisor Mode procedures, it also discards a specified number of bytes from the interrupted program's stack as surplus parameter space. RETT (Return from Trap) is used to return from any trap or interrupt except Maskable interrupts in Vectored mode (Section 6.4). For this, the RETI (Return from Interrupt) instruction is used, which has the additional function of informing any external Interrupt Control Units that interrupt service has completed.

6.4 Maskable Interrupts

Maskable interrupt requests are received on a CPU pin called INT in the current implementation. The input is maskable, and is therefore enabled to generate interrupt requests only while the Processor Status Register I bit is set. The I bit is automatically cleared during service of an INT, NMI or Abort request, and is restored to its original setting upon return from the interrupt service routine via the RETT or RETI instruction.

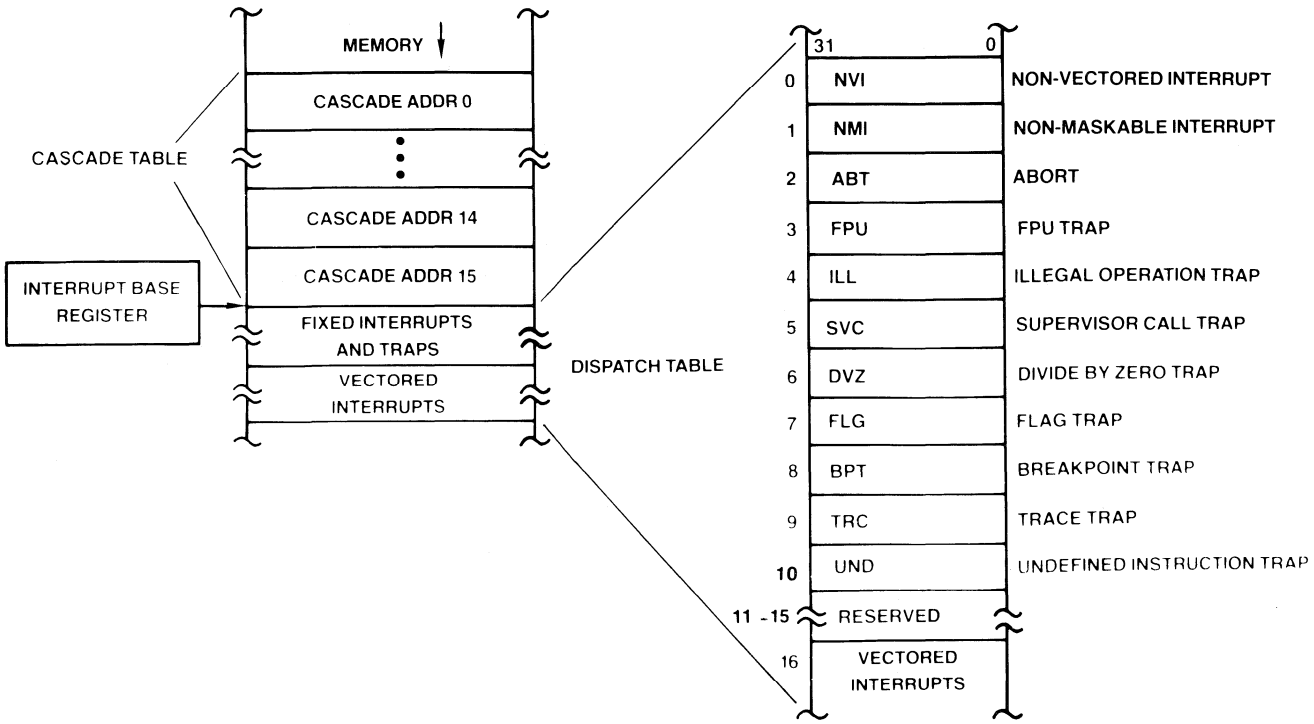


Figure 6-1 Interrupt Dispatch and Cascade Tables

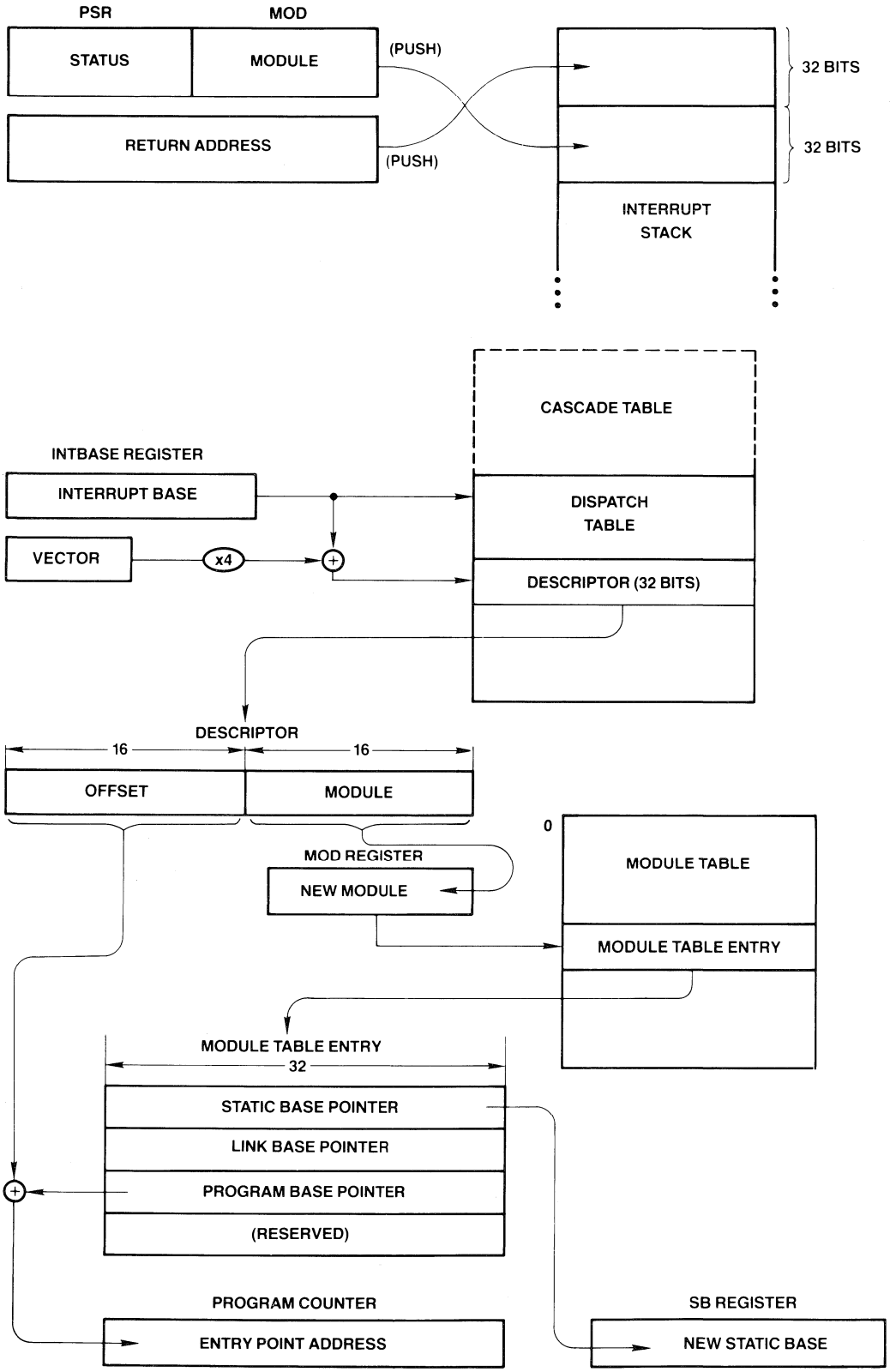


Figure 6-2 Interrupt/ Trap Service Routine Calling Sequence

Maskable interrupt service requested on the INT pin may be performed according to one of two interrupt modes. The interrupt mode is selected via the SETCFG instruction as either Non-Vectored (CFG register bit I = 0) or Vectored (CFG register bit I = 1). The RETT instruction must be used to return from maskable interrupts in Non-Vectored mode, and the RETI instruction must be used to return from maskable interrupts in Vectored mode.

6.4.1 Non-Vectored Mode

In Non-Vectored mode, an interrupt request on the INT pin causes the CPU to read a byte from address 00FFFE00 (Hex), but it ignores any value supplied and uses instead a default vector of zero. This mode is useful for small systems in which hardware interrupt prioritization is unnecessary.

6.4.2 Vectored Mode: Non-Cascaded Case

In Vectored mode, the CPU uses an NS32202 Interrupt Control Unit (ICU) to prioritize up to 16 interrupt requests (see Figure 6-3). Upon receipt of an interrupt request on the INT pin, the CPU reads a 1-byte vector number from address 00FFFE00, asserting a status line to the ICU which indicates Interrupt Acknowledge as the reason for this access. This vector is then used as an index into the Interrupt Dispatch Table in order to find the external procedure descriptor (Section 2.7.3) for the proper interrupt service procedure. The service procedure eventually returns via the Return from Interrupt (RETI) instruction, which performs another one-byte read from address 00FFFE00, this time setting the status line to indicate End of Interrupt, informing the ICU that it may re-prioritize any interrupt requests still pending. The ICU provides the same vector number which was presented previously when the interrupt occurred, and the CPU uses this to determine whether it needs also to inform a cascaded ICU of the end of the service procedure (see Section 6.4.3).

In a system with only one ICU (16 levels of interrupt), the vectors provided must be in the range 0 through 127; that is, they must be positive numbers in eight bits. By providing a negative vector number, an ICU flags the interrupt source as being a cascaded ICU (see Section 6.4.3).

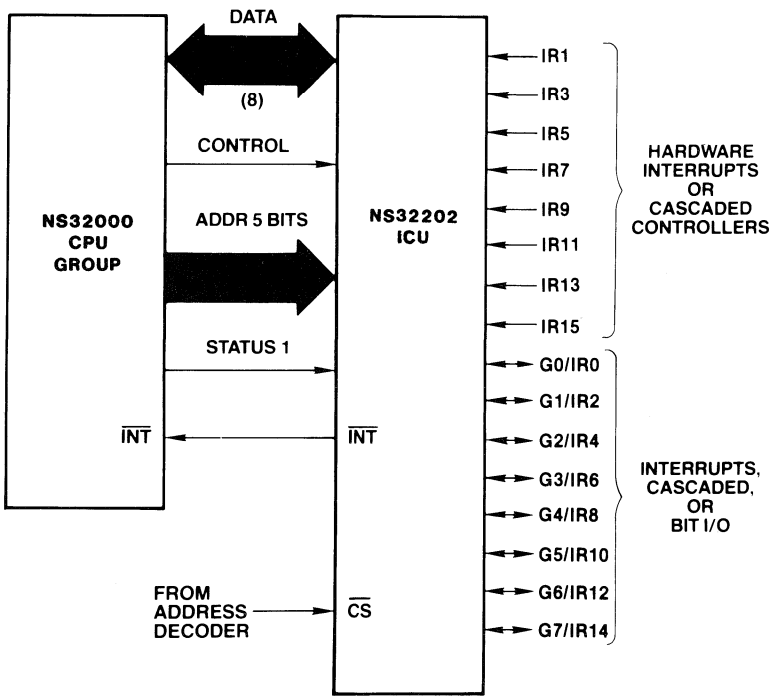


Figure 6-3 Interrupt Control Unit Connections (16 Levels)

6.4.3 Vectored Mode: Cascaded Case

In order to allow up to 256 levels of interrupt, provision is made both in the CPU and in the NS32202 Interrupt Control Unit (ICU) to transparently support cascading. Figure 6-4 shows a typical cascaded configuration. Note that the Interrupt output from a cascaded ICU goes to an Interrupt Request input of the master ICU, which is the only ICU which drives the CPU INT pin.

In a system which uses cascading, two additional initializations must be performed:

1. For each cascaded ICU in the system, the master ICU must be informed of the line number (0 to 15) on which it receives the cascaded requests.

2. A Cascade Table must be established in memory. The Cascade Table is located in a negative direction from the location indicated by the Interrupt Base (INTBASE) register. Its entries contain the 32-bit addresses of the Vector Registers of each of up to 16 cascaded ICUs.

Figure 6-1 illustrates the position of the Cascade Table. To find the Cascade Table entry for any given cascaded ICU, take its Master ICU line number (0 to 15) and subtract 16 from it, giving an index in the range -16 to -1. Multiply this value by 4, and add the resulting negative number to the contents of the INTBASE Register. The 32-bit entry at this address must be set to the address of the Hardware Vector Register of the cascaded ICU. This is referred to as the Cascade Address for that ICU.

Upon receipt of an interrupt request from a cascaded ICU, the master ICU interrupts the CPU and provides the negative Cascade Table index instead of a (positive) vector number. The CPU, seeing the negative value, uses it as an index into the Cascade Table and reads the Cascade Address from the referenced entry. The CPU reads a byte from the address given by the Cascade Address, which it uses as the final vector number. This vector number is interpreted by the CPU as an unsigned integer, and can therefore be in the range 0 through 255.

In returning from a cascaded interrupt, the service procedure executes the Return from Interrupt (RETI) instruction, as it would for any interrupt in Vectored mode. When the CPU performs the End of Interrupt read cycle, the master ICU again provides the negative Cascade Table index. The CPU, seeing a negative value, uses it to find the corresponding Cascade Address from the Cascade Table. Applying this address, it performs an End of Interrupt read cycle, informing the cascaded ICU of the completion of the service routines. The byte read from the cascaded ICU is discarded.

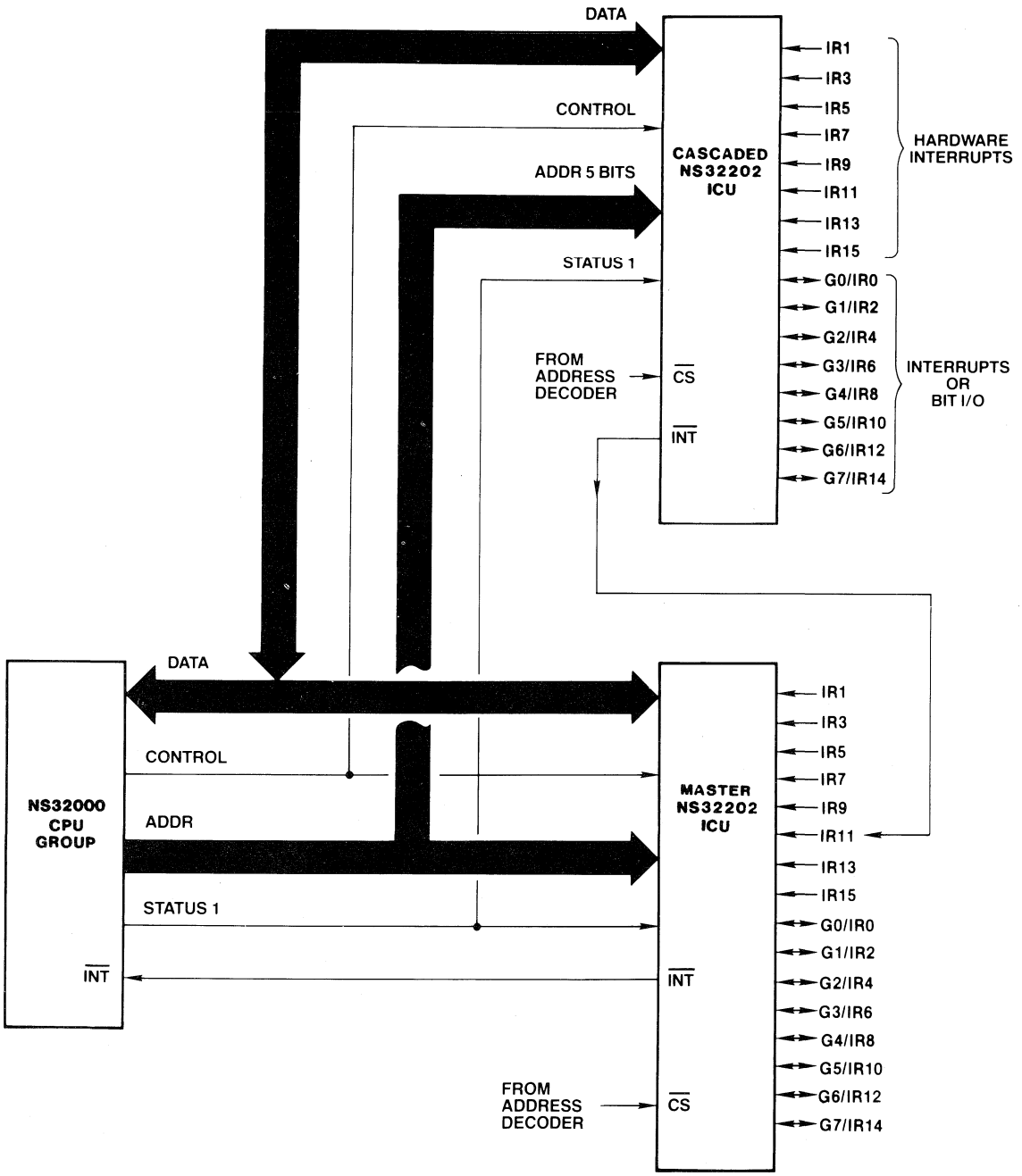


Figure 6-4 Cascaded Interrupt Control Unit Connections

6.5 Non-Maskable Interrupt

The Non-Maskable Interrupt is triggered whenever a falling edge is detected on a CPU pin called NMI. The CPU reads a byte from address 00FFFF00 (Hex) when processing of this interrupt actually begins. Note that this address differs from the address read for Maskable interrupts. Tne vector value used for the Non-Maskable interrupt is 1, regardless of the value read.

The service procedure returns from the Non-Maskable interrupt using the Return from Trap (RETT) instruction. No data transfers occur for interrupt control.

For the full sequence used in processing the Non-Maskable Interrupt, see Section 6.8.1.

6.6 Traps

A trap is an internally generated interrupt request caused as a direct and immediate result of the execution of an instruciton. Traps occurring in a Series 32000 system are:

Trap (ABT): An exceptional condition was detected by memory management.

Trap (FPU): An exceptional condition was detected during the execution of a Floating-Point instruction. In systems incorporating a Custom Slave Processor, this trap can also be caused by an exceptional condition arising there. See Section 3.3 for the conditions which cause this trap.

Trap (ILL): Illegal operation. A privileged operation was attempted while the CPU was in User Mode (PSR U bit = 1). See Section 2.8 for the conditions which cause this trap.

Trap (SVC): The Supervisor Call (SVC) instruction was executed.

Trap (DVZ): An attempt was made to divide an integer by zero. (The FPU trap is used instead for Floating-Point division by zero.) See Section 3.1 for further details.

Trap (FLG): The FLAG instruction detected a "1" in the PSR F bit.

Trap (BPT): The Breakpoint (BPT) instruction was executed.

Trap (TRC): The instruction just completed is being traced. See below.

Trap (UND): An undefined instruction was encountered.

The Return Address pushed by any trap except TRC is the address of the first byte of the instruction during which the trap occurred. Traps do not disable interrupts, as they are not associated with external events.

A special case is the Trace trap, Trap (TRC), which is enabled by setting the T bit in the Processor Status Register (PSR). At the beginning of each instruction, the T bit is copied into the PSR P (Trace "Pending") bit. If the P bit is set at the end of an instruction, the Trace Trap is activated. If any other trap or interrupt request is also pending, its entire service procedure is allowed to complete before the Trace Trap occurs.

Each interrupt and trap sequence handles the P bit for proper tracing, ensuring that one, and only one, Trace Trap will occur per instruction, and that the Return Address pushed during a Trace Trap is always the address of the next instruction to be traced.

6.7 Prioritization

The CPU internally prioritizes simultaneous interrupt and trap requests as follows:

1. Traps other than Trace (Highest priority)
2. Non-Maskable Interrupt
3. Maskable Interrupts
4. Trace Trap (Lowest priority)

Traps at priority level 1 do not occur simultaneously. The first trap occurring is the trap which is serviced.

6.8 Interrupt/Trap Sequences: Detailed Flow

Figure 6-5 defines a common sequence called "Service", which is followed by the CPU in handling all interrupts and traps. Upon detecting any interrupt request or trap condition, the CPU first performs a sequence dependent upon the type of interrupt or trap. This sequence will include pushing the Processor Status Register and establishing a Vector and a Return Address. The CPU then performs the Service sequence.

For the sequence followed in processing either Maskable or Non-maskable interrupts, see Section 6.8.1. For the Abort trap, see Section 6.8.4. For the Trace Trap, see Section 6.8.3 and for all other traps see Section 6.8.2.

6.8.1 Maskable/Non-Maskable Interrupt Sequence

This sequence is performed by the CPU when the NMI pin receives a falling edge, or the INT pin becomes active with the PSR I bit set. The interrupt sequence begins either at the next instruction boundary or, in a String instruction, at the next interruptible point during its execution.

1. If a String instruction is being interrupted and has not yet completed:

   a. Clear the Processor Status Register P bit.

   b. Set "Return Address" to the address of the first byte of the interrupted instruction.

   Otherwise, set "Return Address" to the address of the next instruction.

Service (Vector, Return Address):
1. Push the MOD Register onto the Interrupt Stack as a 16-bit value. (The PSR has already been pushed as a 16-bit value.) 2. Push the Return Address onto the Interrupt Stack as a 32-bit value. 3. Read the 32-bit external procedure descriptor from the Interrupt Dispatch Table: address is Vector*4 + INTBASE. 4. Move the Module field of the descriptor into the MOD Register. 5. Read the new Static Base pointer from the memory address contained in MOD, placing it into the SB Register. 6. Read the Program Base pointer from memory address MOD+8, and add to it the Offset field from the descriptor, placing the result in the Program Counter.

Figure 6-5 Common Interrupt/Trap Service Sequence

2. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits S, U, T, P and I.

3. Push the PSR copy onto the Interrupt Stack as a 16-bit value.

4. If the interrupt is Non-Maskable:

   a. Read a byte from address 00FFFF00 (Hex). Discard the byte read.

   b. Set "Vector" to 1.

   c. Go to Step 9.

5. If the interrupt is Non-Vectored:

   a. Read a byte from address 00FFFE00 (Hex). Discard the byte read.

   b. Set "Vector" to 0.

   c. Go to Step 9.

6. Here the interrupt is being serviced in Vectored mode. Read "Byte" from address 00FFFE00 (Hex).

7. If "Byte" $\geq$ 0, then set "Vector" to "Byte" and go to Step 9.

8. If "Byte" is in the range -16 through -1, then the interrupt source is a cascaded ICU. (More negative values are reserved for future use.) Perform the following:

   a. Read the 32-bit Cascade Address from memory location INTBASE + 4*Byte.

   b. Read "Vector" from the address given by the Cascade Address.

9. Perform Service (Vector, Return Address), Figure 6-5.

6.8.2 Trap Sequence: All Except Trace and Abort

1. Restore the currently selected Stack Pointer to its original contents at the beginning of the trapped instruction.

2. Set "Vector" to the value corresponding to the trap type.

    FPU: Vector = 3.
    ILL: Vector = 4.
    SVC: Vector = 5.
    DVZ: Vector = 6.
    FLG: Vector = 7.
    BPT: Vector = 8.
    UND: Vector = 10.

3. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits S, U, P and T.

4. Push the PSR copy onto the Interrupt Stack as a 16-bit value.

5. Set "Return Address" to the address of the first byte of the trapped instruction.

6. Perform Service (Vector, Return Address), Figure 6-5.

6.8.3 Trace Trap Sequence

1. In the Processor Status Register (PSR), clear the P bit.

2. Copy the PSR into a temporary register, then clear PSR bits S, U and T.

3. Push the PSR copy onto the Interrupt Stack as a 16-bit value.

4. Set "Vector" to 9.

5. Set "Return Address" to the address of the next instruction.

6. Perform Service (Vector, Return Address), Figure 6-5.

6.8.4 Abort Trap Sequence

1. Restore the currently selected Stack Pointer to its original contents at the beginning of the aborted instruction.

2. Clear the PSR P bit.

3. Copy the PSR into a temporary register, then clear PSR bits S, U, T and I.

4. Push the PSR copy onto the Interrupt Stack as a 16-bit value.

5. Set "Vector" to 2.

6. Set "Return Address" to the address of the first byte of the aborted instruction.

7. Perform Service (Vector, Return Address), Figure 6-5.

APPENDIX A

INSTRUCTION SET LISTED BY FUNCTIONAL GROUPS

Instruction	Mnemonic Forms	Index
INTEGER		
Arithmetic		
Add	ADDB, ADDW, ADDD	ADDi
Add Quick	ADDQB, ADDQW, ADDQD	ADDQi
Add with Carry	ADDCB, ADDCW, ADDCD	ADDCi
Subtract	SUBB, SUBW, SUBD	SUBi
Subtract with Carry [Borrow]	SUBCB, SUBCW, SUBCD	SUBCi
Negate	NEGB, NEGW, NEGD	NEGi
Absolute Value	ABSB, ABSW, ABSD	ABSi
Multiply	MULB, MULW, MULD	MULi
Multiply Extended Integer	MEIB, MEIW, MEID	MEIi
Divide	DIVB, DIVW, DIVD	DIVi
Modulus	MODB, MODW, MODD	MODi
Quotient	QUOB, QUOW, QUOD	QUOi
Remainder	REMB, REMW, REMD	REMi
Divide Extended Integer	DEIB, DEIW, DEID	DEIi
Movement and Conversion		
Move	MOVB, MOVW, MOVD	MOVi
Move Quick	MOVQB, MOVQW, MOVQD	MOVQi
Move with Sign-Extension	MOVXBD,MOVXWD,MOVXBW	MOVXii
Move with Zero-Extension	MOVZBD,MOVZWD,MOVZBW	MOVZii
Comparison		
Compare	CMPB, CMPW, CMPD	CMPi
Compare Quick	CMPQB, CMPQW, CMPQD	CMPQi
PACKED DECIMAL		
Add Packed Decimal	ADDPB, ADDPW, ADDPD	ADDPi
Subtract Packed Decimal	SUBPB, SUBPW, SUBPD	SUBPi

Instruction	Mnemonic Forms	Index
FLOATING POINT		
Add Floating	ADDF, ADDL	ADDf
Subtract Floating	SUBF, SUBL	SUBf
Multiply Floating	MULF, MULL	MULf
Divide Floating	DIVF, DIVL	DIVf
Negate Floating	NEGF, NEGL	NEGf
Absolute Value Floating	ABSF, ABSL	ABSf
Compare Floating	CMPF, CMPL	CMPf
Move Floating	MOVF, MOVL	MOVf
Move Long Floating to Floating	MOVLF	
Move Floating to Long Floating	MOVFL	MOVFL
Move Integer to Floating	MOVBF, MOVWF, MOVDF, MOVBL, MOVWL, MOVDL	MOVif
Round Floating to Integer	ROUNDFB, ROUNDFW, ROUNDFD, ROUNDLB, ROUNDLW, ROUNDLD	ROUNDfi
Truncate Floating to Integer	TRUNCFB, TRUNCFW, TRUNCFD, TRUNCLB, TRUNCLW, TRUNCLD	TRUNCfi
Floor Floating to Integer	FLOORFB, FLOORFW, FLOORFD, FLOORLB, FLOORLW, FLOORLD	FLOORfi
Load FSR	LFSR	LFSR
Store FSR	SFSR	SFSR
LOGICAL		
Arithmetic		
Logical AND	ANDB, ANDW, ANDD	ANDi
Logical OR	ORB, ORW, ORD	ORi
Bit Clear	BICB, BICW, BICD	BICi
Exclusive OR	XORB, XORW, XORD	XORi
Complement	COMB, COMW, COMD	COMi
Shift		
Arithmetic Shift	ASHB, ASHW, ASHD	ASHi
Logical Shift	LSHB, LSHW, LSHD	LSHi
Rotate	ROTB, ROTW, ROTD	ROTi
Boolean		
Complement Boolean	NOTB, NOTW, NOTD	NOTi
Save Condition as Boolean	ScondB, ScondW, ScondD	Scondi

Instruction	Mnemonic Forms	Index
BIT		
Test Bit	TBITB, TBITW, TBITD	TBITi
Set Bit	SBITB, SBITW, SBITD, SBITIB, SBITIW, SBITID	SBITi, SBITIi
Clear Bit	CBITB, CBITW, CBITD, CBITIB, CBITIW, CBITID	CBITi, CBITIi
Invert Bit	IBITB, IBITW, IBITD	IBITi
Find First Set Bit	FFSB, FFSW, FFSD	FFSi
Convert to Bit Pointer	CVTP	CVTP
BIT FIELD		
Extract Field	EXTB, EXTW, EXTD	EXTi
Extract Field Short	EXTSB, EXTSW, EXTSD	EXTSi
Insert Field	INSB, INSW, INSD	INSi
Insert Field Short	INSSB, INSSW, INSSD	INSSi
STRING		
Move String	MOVSB, MOVSW, MOVSD	MOVSi
Move String, Translating	MOVST	MOVST
Compare Strings	CMPSB, CMPSW, CMPSD	CMPSi
Compare Strings, Translating	CMPST	CMPST
Skip String	SKPSB, SKPSW, SKPSD	SKPSi
Skip String, Translating	SKPST	SKPST
BLOCK		
Move Multiple	MOVMB, MOVMW, MOVMD	MOVMi
Compare Multiple	CMPMB, CMPMW, CMPMD	CMPMi
ARRAY		
Bounds Check	CHECKB, CHECKW, CHECKD	CHECKi
Calculate Index	INDEXB, INDEXW, INDEXD	INDEXi

Instruction	Mnemonic Forms	Index
PROCESSOR CONTROL		
Branches		
Jump	JUMP	JUMP
Conditional Branch	Bcond	Bcond
Unconditional Branch	BR	BR
Case Branch (Multiway)	CASEB, CASEW, CASED	CASEi
Add, Compare and Branch	ACBB, ACBW, ACBD	ACBi
Local Procedure Calls/Returns		
Jump to Subroutine	JSR	JSR
Branch to Subroutine	BSR	BSR
Return from Subroutine	RET	RET
External Procedure Calls/Returns		
Call External Procedure	CXP	CXP
Call External Procedure with Descriptor	CXPD	CXPD
Return from External Procedure	RXP	RXP
Explicit Trap Instructions		
Breakpoint Trap	BPT	BPT
Trap on Flag (conditional)	FLAG	FLAG
Supervisor Call Trap	SVC	SVC
Trap/Interrupt Returns		
Return from Trap*	RETT	RETT
Return from Interrupt*	RETI	RETI

* Privileged instruction.

Instruction	Mnemonic Forms	Index
PROCESSOR SERVICE		
Effective Address		
Calculate Effective Address	ADDR	ADDR
Load External Procedure Descriptor (alternate mnemonic for ADDR)	LXPD	LXPD
Context Instructions		
Save General Purpose Registers	SAVE	SAVE
Restore General Purpose Registers	RESTORE	RESTORE
Enter New Procedure Context	ENTER	ENTER
Exit Procedure Context	EXIT	EXIT
Register/Stack Manipulation		
Adjust Stack Pointer	ADJSPB, ADJSPW, ADJSPD	ADJSPi
Bit Clear in PSR*	BICPSRB, BICPSRW	BICPSRB BICPSRW
Bit Set in PSR*	BISPSRB, BISPSRW	BISPSRB BISPSRW
Load Processor Register*	LPRB, LPRW, LPRD	LPRi
Store Processor Register*	SPRB, SPRW, SPRD	SPRi
Set Configuration Register*	SETCFG	SETCFG
Miscellaneous		
No Operation	NOP	NOP
Wait for Interrupt	WAIT	WAIT
Diagnose	DIA	DIA
MEMORY MANAGEMENT		
Load Memory Management Register	LMR	LMR
Store Memory Management Register	SMR	SMR
Validate Address for Reading	RDVAL	RDVAL
Validate Address for Writing	WRVAL	WRVAL
Move Value from Supervisor to User Space	MOVSUB, MOVSUW, MOVSUD	MOVSUi
Move Value from User to Supervisor Space	MOVUSB, MOVUSW, MOVUSD	MOVUSi

* Privileged, or having privileged forms.

APPENDIX B

NS32016 INSTRUCTION EXECUTION TIMES

B.1 Assumptions

The entire instruction, with all displacements and immediate operands, is assumed to be present in the instruction queue when needed.

Interference from instruction prefetches, which is very dependent upon the preceding instruction(s), is ignored. This assumption will tend to affect the timing estimate in an optimistic direction.

It is assumed that all memory operand transfers are completed before the next instruction begins execution. In the case of an operand of access class rmw in memory, this is pessimistic, as the Write transfer occurs in parallel with the execution of the next instruction.

It is assumed that there is no overlap between the fetch of an operand and the following sequences of microcode. This is pessimistic, as the fetch of Operand A will generally occur in parallel with the effective address calculation of Operand B, and the fetch of Operand B will occur in parallel with the execution phase of the instruction. See Section 4.3 for the definition of operands A and B.

Where possible, the values of operands are taken into consideration when they affect instruction timing, and a range of times is given. Where this is not done, the worst case is assumed.

B.2 Definitions

TEA - The time required to calculate an operand's Effective Address. For a Register or Immediate operand, this includes the fetch of that operand.

TMMU - The extra clock cycle required for translation of memory addresses if an MMU is present.

TOPB - The time needed to read or write a memory byte.

TOPW - The time needed to read or write a memory word.

TOPD - The time needed to read or write a memory double-word.

TOPi - The time needed to read or write a memory operand, where the operand size is given by the operation length of the instruction. It is always equivalent to either TOPB, TOPW or TOPD.

TCY - Internal processing overhead, in clock cycles.

L - Internal processing whose duration depends on the operation length (Section 4.1). The number of clock cycles is derived by multiplying this value by the number of bytes in the operation length.

B.3 Equations

TMMU - If an MMU is present then TMMU=1
else TMMU=0

TOPB - If operand is in a register or is immediate then TOPB=0
else TOPB = 3 + TMMU

TOPW - If operand is in a register or is immediate then TOPW=0
else if word-aligned (even address) then TOPW = 3 + TMMU
else TOPW = 7 + 2 * TMMU

TOPD - If operand is in a register or is immediate then TOPD=0
else if word-aligned (even address) then TOPD = 7 + 2 * TMMU
else TOPD = 11 + 3 * TMMU

TOPi - If operand is in a register or is immediate then TOPi=0
else if i=byte then TOPi = TOPB
else if i=word then TOPi = TOPW
else (i=double-word) then TOPi = TOPD

TCY - TCY = 1

L - If i (operation length) = byte then L = 1
else if i = word then L = 2
else (i = double-word) L = 4

TEA - If REGISTER addressing then TEA = 2
if IMMEDIATE or ABSOLUTE addressing then TEA = 4
if REGISTER RELATIVE or MEMORY SPACE addressing then TEA = 5
if MEMORY RELATIVE addressing then TEA = 7 + TOPD
if TOP OF STACK addressing then
if access class = write then TEA = 4
if access class = read then TEA = 2
else TEA = 3
if EXTERNAL addressing then TEA = 11 + 2 * TOPD
if SCALED INDEXED addressing then TEA = TI1 + TI2

where TI1 depends on scale factor:
if byte indexing TI1 = 5
if word indexing TI1 = 7
if double-word indexing TI1 = 8
if quad-word indexing TI1 = 10

and TI2 = TEA of the basemode except:
if basemode is REGISTER then TI2 = 5
if basemode is TOP OF STACK then TI2 = 4

B.4 Calculation of Total Execution Time (TEX)

TEX is obtained by performing the following steps:

1. Find the desired instruction in the table.

2. Calculate the values for TEA, TOPB, etc. using the numbers in the table and the equations given on the preceding page.

3. The result derived by adding together these values is the execution time (TEX) in clock cycles.

B.5 Notes on Table Use

Values in the TEA column indicate the number of effective addresses to be calculated. If the value in this column is less than the number of general operands in the instruction, this is because one or both operands are in registers and that instruction has an optimized form which eliminates TEA for such operands.

In the L column, multiply the entry by the operation length in bytes (1, 2 or 4).

In the TCY column, special notations sometimes appear:

n1-n2 means n1 minimum, n2 maximum.

n1%n2 means that the instruction flushes the instruction queue after n1 clock cycles and nonsequentially fetches the next instruction. The value n2, indicating the total number of clock cycles in internally executing the instruction (including n1), is not generally useful. The most accurate technique for determining such timing depends on the size and alignment of the Basic Instruction portion of the next instruction, plus Index Bytes. If this portion can be read in one memory cycle, then the execution time is n1+10 (including the memory cycle). If more memory cycles are required, the value is n1+5+4*m, where m is the number of memory cycles required.

In the Notes column, notations held within angle brackets <> indicate alternatives in the form of the instruction which affect the execution time. A table entry which is affected by the form of the instruction may have multiple values, separated by slashes, corresponding to the alternatives. The notations are:

<M>	Memory form
<R>	Register form
<MM>	Memory-to-Memory form
<RM>	Register-to-Memory form
<MR>	Memory-to-Register form
<RR>	Register-to-Register form
x	either Register or Memory

B.6 Example of Table Usage

Calculate TEX for the instruction:

```
CMPW  R0,TOS
```

Operand A is in a register; Operand B is in memory. This means that we must use the table values corresponding to the <xM> case as given in the Notes column (<xM> meaning "anything to memory").

Only the TEA, TOPi and TCY columns have values assigned for the CMPi instruction. Therefore, they are the only ones that need to be calculated to find TEX. The blank columns are irrelevant to this instruction.

The TEA column contains 2 for the <xM> case. This means that effective address times have to be calculated for both operands. (For the <MR> case, the Register operand would have required no TEA time, therefore only the Memory operand TEA would have been necessary.) From the equations:

```
TEA (Register mode)  =  2,
TEA (Top of Stack mode, read access class)  =  2,

Total TEA = 2+2 = 4.
```

The TOPi column represents potential operand transfers to or from memory. For a Compare instruction, each operand is read once, for a total of two operand transfers.

```
TOPi (Word, Register) = 0,
TOPi (Word, TOS) = TOPW = 3  (assuming aligned, no MMU)
Total TOPi = 3
```

TCY is the time required for internal operation within the CPU. The TCY value for this case is 3.

TEX = TEA + TOPi + TCY = 4 + 3 + 3 = 10 machine cycles.

If the CPU is running at 10 MHz then a machine cycle (clock cycle) is 100 nsec. Therefore, this instruction would take 10x100 nsec, or 1.0 microseconds, to execute.

Table B-1 Basic and Memory Management Instructions

This table does not include the timings for Floating-Point instructions. See Section B.7 and Table B-2.

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
ABSi	2	-	-	-	2	9/8	-	src<0 / src>=0
ACBi	1	-	-	-	2	16/15%20	-	<M>, no branch / branch
ACBi	-	-	-	-	-	18/17%22	-	<R>, no branch / branch
ADDi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
ADDCi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
ADDPi	2	-	-	-	3	16/18	-	no carry / carry
ADDQi	1/0	-	-	-	2/0	6/4	-	<M>/<R>
ADDR	2/1	-	-	1/0	-	2/3	-	<xM>/<xR>
ADJSPi	1	-	-	-	1	6	-	
ANDi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
ASHi	2	1	-	-	2	14-45	-	
Bcond	-	-	-	-	-	7/6%10	-	no branch / branch
BICi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
BICPSRB	1	1	-	-	-	18%22	-	
BICPSRW	1	-	1	-	-	30%34	-	

Table B-1 (Cont.)

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
BISPSRB	1	1	-	-	-	18%22	-	
BISPSRW	1	-	1	-	-	30%34	-	
BPT	-	-	4	3	-	40	-	
BR	1	-	-	-	1	4%9	-	
BSR	-	-	-	1	-	6%16	-	
CASEi	1	-	-	-	1	4%9	-	
CBITi	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
CBITIi	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
CHECKi	2	-	-	-	3	7/10/11	-	high / low / ok
CMPi	2/1/0	-	-	-	2/1/0	3	-	<xM>/<MR>/<RR>
CMPMi	2	-	-	-	2*n	9*n+24	-	n = # of elements in block
CMPQi	1/0	-	-	-	1/0	3	-	<M>/<R>
CMPSi	-	-	-	-	2*n	35*n+53	-	n = # of elements, not Translated
CMPST	-	n	-	-	2*n	38*n+53	-	Translated
COMi	2	-	-	-	2	7	-	
CVTP	2	-	-	1	-	7	-	
CXP	-	-	3	4	-	16%21	-	

Table B-1 (Cont.)

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
CXPD	1	-	3	3	-	13%18	-	
DEIi	2/1	-	-	-	5/1	38/31	16	<xM>/<xR>
DIA	-	-	-	-	-	3%7	-	
DIVi	2	-	-	-	3	58-68	16	
ENTER	-	-	-	n+1	-	4*n+18	-	n = # of general registers saved
EXIT	-	-	-	n+1	-	5*n+17	-	n = # of general registers restored
EXTi	2	-	-	1	1	19-29	-	field in memory
EXTi	2	-	-	-	1	17-51	-	field in register
EXTSi	2	-	-	1	1	26-36	-	
FFSi	2	2	-	-	1	24-28	24	
FLAG	-	-	0/4	0/3	-	6/44	-	no trap / trap
IBITi	2/1	2/0	-	-	1	17/9	-	<xM>/<xR>
INDEXi	2	-	-	-	2	25	16	
INSi	2	-	-	2	1	29-39	-	field in memory
INSi	1	-	-	-	1	28-96	-	field in register
INSSi	2	-	-	2	1	39-49	-	
JSR	1	-	-	1	1	5%15	-	

Table B-1 (Cont.)

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
JUMP	1	-	-	-	-	2%6	-	
LMR	1	-	-	-	1	30%34	-	
LPRi	1	-	-	-	1	19-33	-	
LSHi	2	1	-	-	2	14-45	-	
MEIi	2	-	-	-	4	23	16	
MODi	2	-	-	-	3	54-73	16	
MOVi	2/1/0	-	-	-	2/1/0	1/3/3	-	<xM>/<MR>/<RR>
MOVMi	2	-	-	-	2*n	3*n+20	-	n = # of elements in block
MOVQi	1/0	-	-	-	1/0	2/3	-	<M>/<R>
MOVSi	-	-	-	-	2*n	13*n+18	-	n = # of elements, no options
MOVSi	-	-	-	-	2*n	24*n+54	-	B, W and/or U option in effect
MOVST	-	n	-	-	2*n	27*n+54	-	Translated
MOVSUi	2	-	-	-	2	33%37	-	
MOVUSi	2	-	-	-	2	33%37	-	
MOVXBD	2	1	-	1	-	6	-	
MOVXBW	2	1	1	-	-	6	-	
MOVXWD	2	-	1	1	-	6	-	

Table B-1 (Cont.)

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
MOVZBD	2	1	-	1	-	5	-	
MOVZBW	2	1	1	-	-	5	-	
MOVZWD	2	-	1	1	-	5	-	
MULi	2	-	-	-	3	15	16	
NEGi	2	-	-	-	2	5	-	
NOP	-	-	-	-	-	3	-	
NOTi	2	-	-	-	2	5	-	
ORi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
QUOi	2	-	-	-	3	49-55	16	
RDVAL	1	1	-	-	-	21	-	
REMi	2	-	-	-	3	57-62	16	
RESTORE	-	-	-	n	-	5*n+12	-	n = # of general registers restored
RET	-	-	-	1	-	2%8	-	
RETI	-	1	3	3	-	39%45	-	
RETT	-	-	2	2	-	35%41	-	
ROTi	2	1	-	-	2	14-45	-	
RXP	-	-	1	2	-	2%6	-	

Table B-1 (Cont.)

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
Scondi	1	-	-	-	1	9/10	-	False / True
SAVE	-	-	-	n	-	4*n+13	-	n = # of general registers saved
SBITi	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
SBITIi	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
SETCFG	-	-	-	-	-	15	-	
SKPSi	-	-	-	-	n	27*n+51	-	n = # of elements, not Translated
SKPST	-	n	-	-	n	30*n+51	-	Translated
SMR	1	-	-	-	1	25	-	
SPRi	1	-	-	-	1	21-27	-	
SUBi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
SUBCi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
SUBPi	2	-	-	-	3	16/18	-	no carry / carry
SVC	-	-	4	3	-	40	-	
TBITi	2/1	1/0	-	-	1	14/4	-	<xM>/<xR>
WAIT	-	-	-	-	-	6-?	-	? = until an interrupt/reset
WRVAL	1	1	-	-	-	21	-	
XORi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>

B.7 Floating-Point Execution Times

Table B-2 gives execution timing information for Floating-Point instructions. Some additional timing definitions are used, as given below.

f - The floating-point operation length.

Standard Floating (32 bits): f=1
Long Floating (64 bits): f=2

Tf - The time required to transfer 32 bits of a floating-point value to or from the NS32081 Floating-Point Unit.

Tf = 4 always.

Ti - The time required to transfer an integer value to or from the NS16081 Floating-Point Unit.

Byte: Ti = 2
Word: Ti = 2
Double-Word: Ti = 4

Table B-2 Floating-Point Instruction Execution Times

INSTRUCTION	CASE	TEA	TOPD	TOPi	Ti	Tf	TCY
MOVf	<MM>	2	2f	-	-	2f	23
	<RR>	-	-	-	-	-	27
	<MR>	1	f	-	-	f	23
	<RM>	-	f	-	-	f	27
ADDf, SUBf	<MM>	2	3f	-	-	3f	70
	<RR>	-	-	-	-	-	74
	<MR>	1	f	-	-	f	70
	<RM>	1	2f	-	-	2f	70
MULf	<MM>	2	3f	-	-	3f	30+14f
	<RR>	-	-	-	-	-	34+14f
	<MR>	1	f	-	-	f	30+14f
	<RM>	1	2f	-	-	2f	30+14f
DIVf	<MM>	2	3f	-	-	3f	55+30f
	<RR>	-	-	-	-	-	59+30f
	<MR>	1	f	-	-	f	55+30f
	<RM>	1	2f	-	-	2f	55+30f
ABSf, NEGf	<MM>	1	2f	-	-	2f	20
	<RR>	-	-	-	-	-	24
	<MR>	1	f	-	-	f	20
	<RM>	-	f	-	-	f	24
CMPf	<MM>	2	2f	-	-	2f	45
	<RR>	-	-	-	-	-	49
	<MR>	1	f	-	-	f	45
	<RM>	1	f	-	-	f	45
MOVLF	<MM>	1	3	-	-	3	23
	<RR>	-	-	-	-	-	27
	<MR>	1	2	-	-	2	23
	<RM>	-	1	-	-	1	27
MOVFL	<MM>	1	3	-	-	3	22
	<RR>	-	-	-	-	-	26
	<MR>	1	1	-	-	1	22
	<RM>	-	2	-	-	2	26
MOVif	<MM>	1	f	1	1	f	53
	<MR>	1	-	1	1	-	53
ROUNDfi, TRUNCfi, FLOORfi	<MM>	1	f	1	1	f	53
	<RM>	-	-	1	1	-	66
SFSR	<M>	-	1	-	-	1	13
LFSR	<M>	1	1	-	-	1	18