CONTENTS

Chapter			Page
1	INTRODUCTION		1-1
2	PROGRAMMING MODEL		2-1
	2.1	GENERAL REGISTERS	2-2
	2.2	DEDICATED REGISTERS	2-2
	2.3	CONFIGURATION REGISTER (CFG)	2-10
	2.4	FLOATING-POINT REGISTERS	2-11
		2.4.1 Floating-Point Data Registers	2-11
		2.4.2 Floating-Point Status Register (FSR)	2-12
	2.5	MEMORY MANAGEMENT REGISTERS	2-15
	2.6	MEMORY ORGANIZATION	2-17
		2.6.1 Addressing	2-17
		2.6.2 Memory Operand Formats	2-17
		2.6.3 Data Alignment	2-19
	2.7	DEDICATED MEMORY AREAS	2-19
		2.7.1 User and Interrupt Stacks	2-20
		2.7.2 Module Table	2-21
		2.7.3 Link Tables	2-23
		2.7.4 Interrupt Dispatch Table and Cascade Table	2-24
		2.7.5 Input and Output	2-24
	2.8	PRIVILEGE STATES AND PROTECTION	2-25
3	INSTRUCTIONS AND DATA TYPES		3-1
	3.1	INTEGER INSTRUCTIONS	3-2
	3.2	PACKED DECIMAL INSTRUCTIONS	3-7
	3.3	FLOATING-POINT INSTRUCTIONS	3-9
		3.3.1 Floating-Point Operand Formats	3-10
		3.3.2 Normalized Numbers	3-11
		3.3.3 Zero	3-12
		3.3.4 Reserved Operands	3-13
		3.3.5 Integers	3-13
		3.3.6 Memory Representations	3-13
		3.3.7 Floating-Point Traps	3-14
	3.4	LOGICAL INSTRUCTIONS	3-16
	3.5	BIT INSTRUCTIONS	3-18
	3.6	BIT FIELD INSTRUCTIONS	3-21
	3.7	STRING INSTRUCTIONS	3-24
	3.8	BLOCK INSTRUCTIONS	3-29
	3.9	ARRAY INSTRUCTIONS	3-31
	3.10	PROCESSOR CONTROL INSTRUCTIONS	3-33
	3.11	PROCESSOR SERVICE INSTRUCTIONS	3-35
	3.12	MEMORY MANAGEMENT INSTRUCTIONS	3-37
	3.13	CUSTOM INSTRUCTIONS	3-38
4	INSTRUCTION OPTIONS AND CONSTRUCTION		4-1
	4.1	SYNTAX PRESENTATION	4-2
	4.2	OPERAND ATTRIBUTES	4-3

Chapter Page

4.2.1 Access Classes 4-4
4.2.2 Length Attributes 4-6
4.2.2.1 Integer Length Attributes 4-7
4.2.2.2 Floating-Point Length Attributes 4-8
4.2.3 Implied Operand Attributes 4-9
4.3 BINARY INSTRUCTION FORMAT 4-10
4.3.1 Basic Instruction 4-12
4.3.1.1 Operation Code Fields 4-12
4.3.1.2 Operation Length Fields: i and f 4-12
4.3.1.3 General Addressing Mode Fields: gen 4-12
4.3.1.4 Implied Operand Fields: reg,quick,short .. 4-13
4.3.2 Extension Fields 4-13
4.3.2.1 Index Bytes 4-13
4.3.2.2 Addressing Extensions 4-13
4.3.2.3 Implied Operand Extensions: imm, disp 4-14
4.4 NS16000 ADDRESSING MODES 4-15
4.4.1 Register Modes 4-17
4.4.2 Register Relative Modes 4-19
4.4.3 Memory Relative Modes 4-20
4.4.4 Immediate Mode 4-21
4.4.5 Absolute Mode 4-22
4.4.6 External Mode 4-23
4.4.7 Top of Stack Mode 4-24
4.4.8 Memory Space Modes 4-25
4.4.9 Scaled Indexing 4-26
4.5 CONSTRUCTING COMPLETE BINARY INSTRUCTIONS: SOME EXAMPLES ... 4-28

5 NS16000 INSTRUCTION SET 5-1

5.1 INSTRUCTION EXAMPLES 5-3
5.1.1 Coding Examples 5-3
5.1.2 Action Examples 5-3
5.1.3 Operand Presentation Format 5-5
5.2 INSTRUCTION DEFINITIONS 5-7

6 EXCEPTION PROCESSING 6-1

6.1 RESET 6-1
6.2 GENERAL INTERRUPT/TRAP SEQUENCE 6-1
6.3 INTERRUPT/TRAP RETURN 6-2
6.4 MASKABLE INTERRUPTS 6-2
6.4.1 Non-Vectored Mode 6-5
6.4.2 Vectored Mode: Non-Cascaded Case 6-5
6.4.3 Vectored Mode: Cascaded Case 6-7
6.5 NON-MASKABLE INTERRUPT 6-9
6.6 TRAPS 6-9
6.7 PRIORITIZATION 6-11

CONTENTS (Cont.)

Chapter Page

6.8 INTERRUPT/TRAP SEQUENCES: DETAILED FLOW 6-11
6.8.1 Maskable/Non-Maskable Interrupt Sequence 6-11
6.8.2 Trap Sequence: All Except Trace and Abort 6-14
6.8.3 Trace Trap Sequence 6-14
6.8.4 Abort Trap Sequence 6-15

Appendix

A INSTRUCTION SET LISTED BY FUNCTIONAL GROUPS A-1

B NS16032 INSTRUCTION EXECUTION TIMES B-1

B.1 ASSUMPTIONS B-1
B.2 DEFINITIONS B-1
B.3 EQUATIONS B-2
B.4 CALCULATION OF TOTAL EXECUTION TIME (TEX) B-3
B.5 NOTES ON TABLE USE B-3
B.6 EXAMPLE OF TABLE USAGE B-4
B.7 FLOATING-POINT EXECUTION TIMES B-12

ILLUSTRATIONS

Figure Page

2-1 NS16000 Register Set 2-3
2-2 Processor Status Register 2-6
2-3 Floating-Point Status Register 2-12
2-4 Module Descripter Format 2-21
2-5 Sample Link Table 2-23
3-1 Floating-Point Operand Formats 3-10
4-1 General Format 4-11
5-1 Typical Instruction Definition 5-2
5-2 Typical Instruction Example 5-4
6-1 Interrupt Dispatch and Cascade Tables 6-3
6-2 Interrupt/Trap Service Routine Calling Sequence 6-4
6-3 Interrupt Control Unit Connections (16 Levels) 6-6
6-4 Cascaded Interrupt Control Unit Connections 6-8
6-5 Common Interrupt/Trap Service Sequence 6-12

TABLES

Table		Page
2-1	PRIVILEGED INSTRUCTIONS	2-26
3-1	SAMPLE F FIELDS	3-10
3-2	SAMPLE E FIELDS	3-11
3-3	NORMALIZED FLOATING-POINT RANGES	3-12
3-4	EXECUTION SEQUENCES	3-28
3-5	ROW MAJOR ORDERING	3-32
3-6	COLUMN MAJOR ORDERING	3-32
4-1	ADDRESSING MODE ACTIONS VS. ACCESS CLASS	4-5
4-2	NS16000 ADDRESSING MODES	4-16
B-1	BASIC AND MEMORY MANAGEMENT INSTRUCTIONS	B-5
B-2	FLOATING-POINT INSTRUCTION EXECUTION TIMES	B-13

Chapter 1

INTRODUCTION

This document is a revised definition of the NS16000 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 NS16082 MMU data sheet.

The chapters appearing in this release are:

1. INTRODUCTION

2. PROGRAMMING MODEL
   Definitions of the NS16000 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 NS16000 addressing modes and the construction of instructions in assembly language and binary.

5. INSTRUCTION SET
   Individual definitions of the NS16000 instructions, organized alphabetically by mnemonic.

6. EXCEPTION PROCESSING
   Definitions of the NS16000 interrupt and trap structure, including the response to a Reset.

Appendices:

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

A note on terminology

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 NS16000 family.

Chapter 2

PROGRAMMING MODEL

This chapter defines the programming model (resources visible to the programmer) presented by the NS16000 architecture. More specifically, this chapter presents the NS16000 register set, memory organization, and the functions of dedicated memory areas used by NS16000 hardware. Also presented here is the mechanism used to protect privileged portions of the programming model.

This chapter is organized as follows:

Topic	**Section**
General Registers	2.1
Dedicated Registers	2.2
Configuration Register	2.3
Floating-Point Registers	2.4
Memory Management Registers	2.5
Memory Organization	2.6
Dedicated Memory Areas	2.7
Privilege States and Protection	2.8

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 NS16000 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.

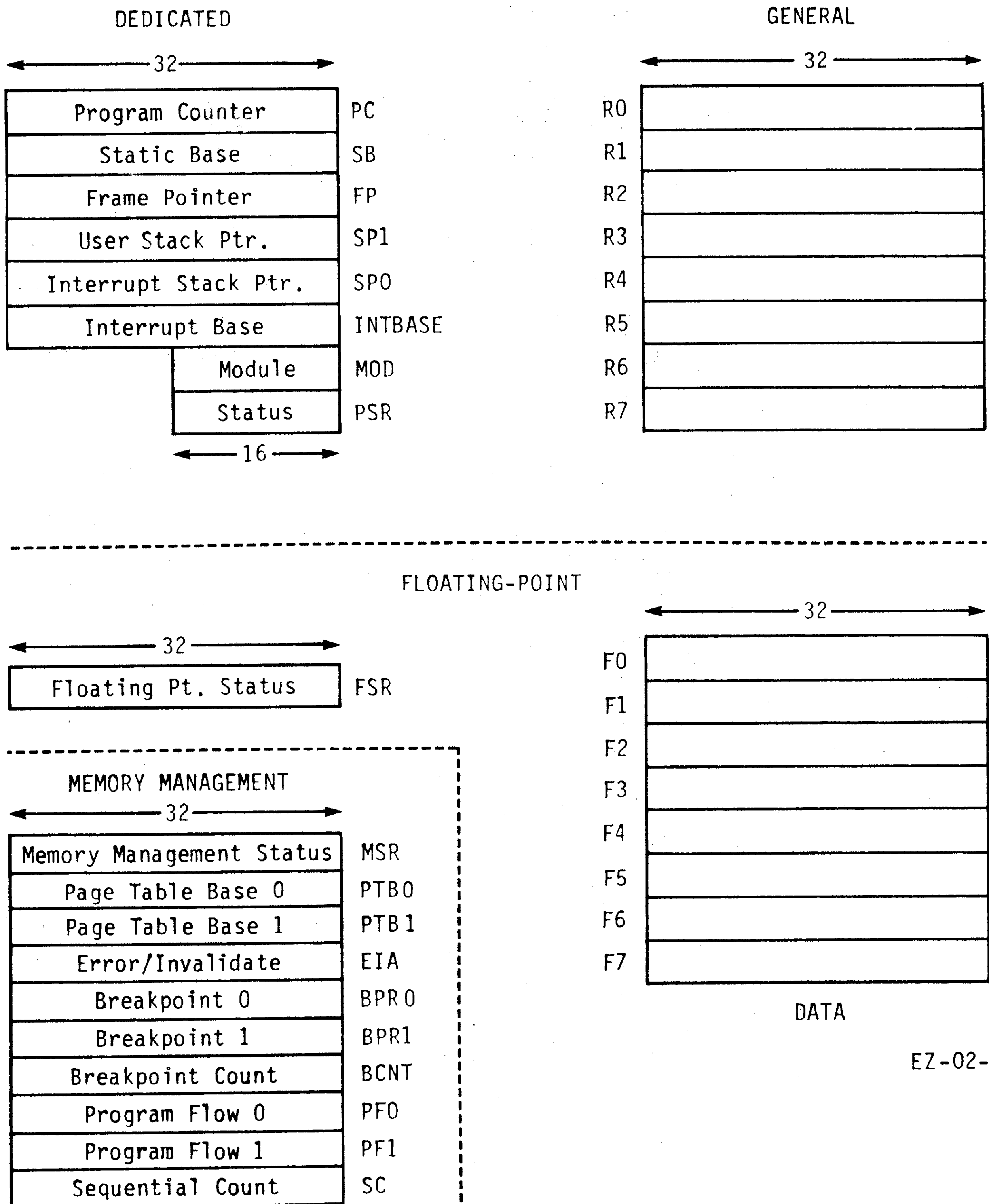


Figure 2-1: NS16000 Register Set

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).

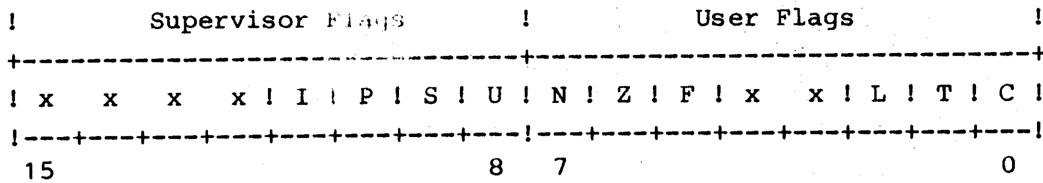
INTBASE — The Interrupt Base register contains the base address of the Interrupt Dispatch Table. This is a vector table which contains the descriptors of the trap and interrupt service procedures. See Chapter 6 for details of trap and interrupt handling. In memory-managed systems, INTBASE must always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

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 below). 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 below) 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).

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 NS16000 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

```
+---+---+---+---+
! C ! M ! F ! I !
+---+---+---+---+
```

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 NS16202 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 NS16081 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. (Note: The trap mechanism employed by the NS16000 architecture allows software to intercept this trap and fully emulate the functions of the NS1608[illegible]

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 NS16082 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 NS16081 Floating-Point Unit or by software emulation). See Figure 2-1. There are nine Floating-Point registers, consisting of:

Floating-Point Data Registers (F0-F7)
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.

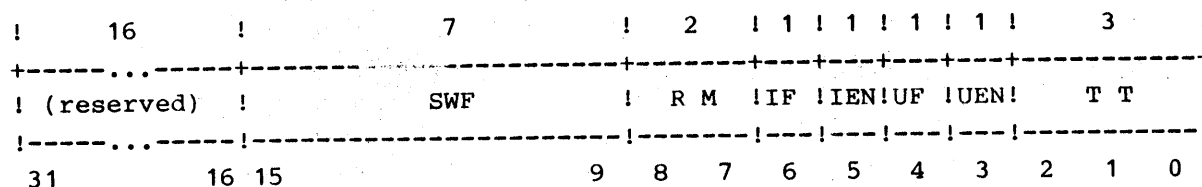


Figure 2-3: Floating-Point Status Register

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 into 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 given below.

TT Trap Type: bits 0-2. This 3-bit field indicates the reason for any Trap (FPU) which is caused by a floating-point instruction. It is cleared by writing zeroes into it with the Load FSR (LFSR) instruction or by a Reset (Chapter 6). The conditions causing Trap (FPU) are indicated as given below. These conditions are defined in Section 3.3.7.

000	No trap requested.
001	Underflow
010	Overflow
011	Divide by Zero
100	Illegal Instruction
101	Invalid Operation
110	Inexact Result
111	(Reserved for future use.)

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 NS16000 memory management option. These registers are currently implemented in the NS16082 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), consisting of:

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

2.6 Memory Organization

The NS16000 architecture supports a memory addressing space of 4 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.

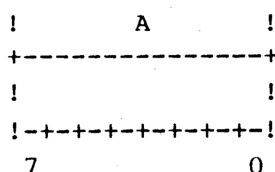
NOTES:

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

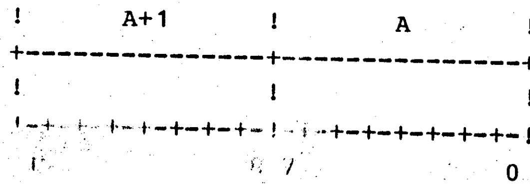
The basic storage unit is the byte. A byte holds eight bits of data and has the following form:



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.

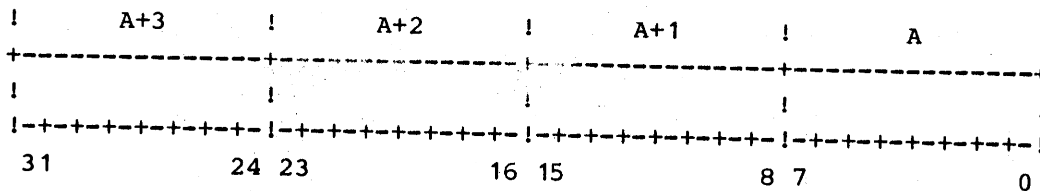
A 16-bit value is called a word. It is held in memory as a pair of contiguous bytes.



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.

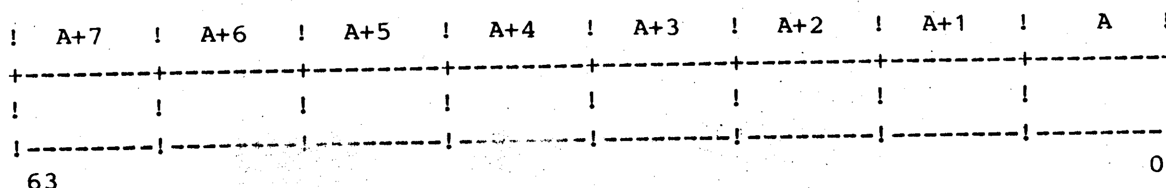
A 32-bit value is called a double-word. It is held in memory as four contiguous bytes. A double-word can hold either a 32-bit integer or a single-precision floating-point value.



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 NS16000 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

An NS16000-based system will make use of certain designated memory areas for the following purposes:

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

2.7.1 User and Int[illegible]

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 Point[illegible] [illegible] address of the top of the Interrupt Stack. At any time, [illegible] 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 St[illegible] addressi[illegible] (Section 4.4.7), which performs an automatic push or pop, as appropriat[illegible]. 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-[illegible]ed systems, the Interrupt stack must always be available in phy[illegible] memory. On occurrence of an interrupt or trap, the contents [illegible] Interrupt stack pointer are treated as a Supervisor-Mode virtual [illegible]ss.

2.7.2 Module Table

The NS16000 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. An NS16000-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.

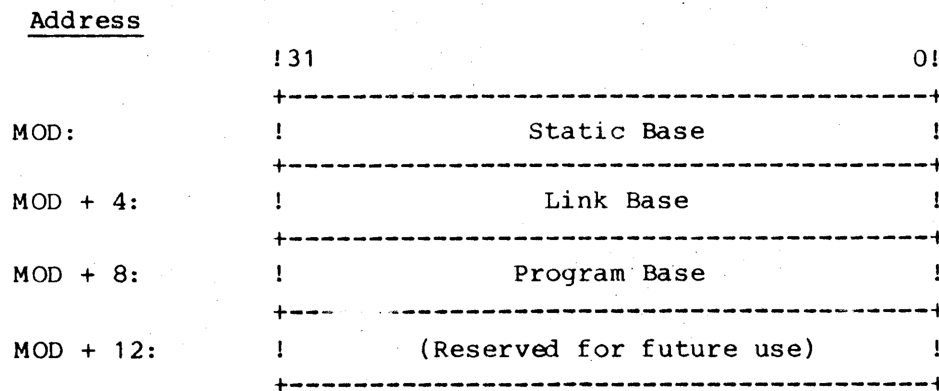


Figure 2-4: Module Descriptor Format

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 [illegible] entirely contained within the first 64K bytes [illegible] 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 <u>procedure descriptor</u> 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).

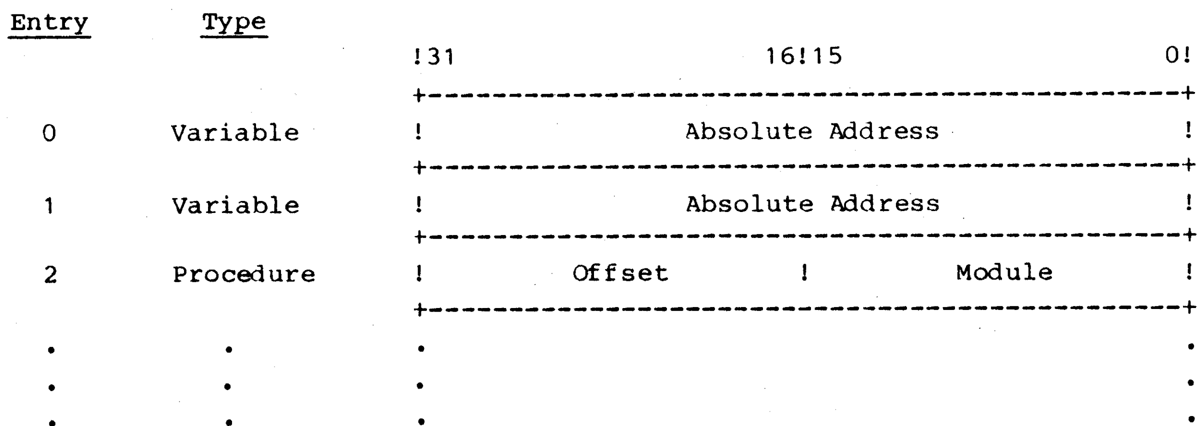


Figure 2-5: Sample Link Table

2.7.4 Interrupt Dispatch Table and Cascade Table

The NS16000 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. An NS16000-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 NS16000-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 NS16000 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 NS16000 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 entire PSR)	LPRi
Store Processor Register	(if INTBASE or entire 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 NS16000 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

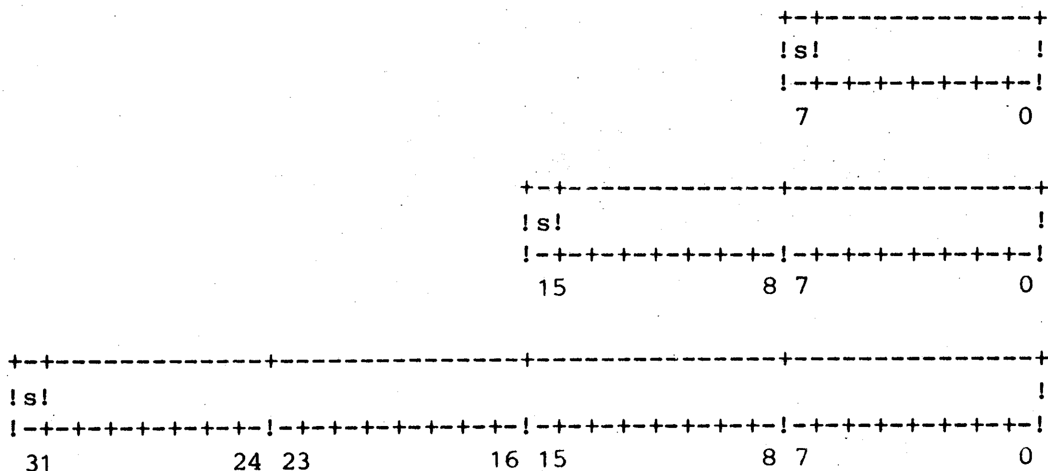
Integer instructions operate on byte, word, and double-word integer operands. The following is a list of the Integer instructions:

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.

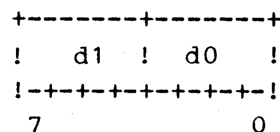
See Chapter 6 for details of trap handling.

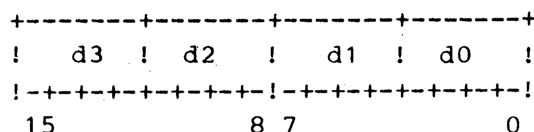
3.2 Packed Decimal Instructions

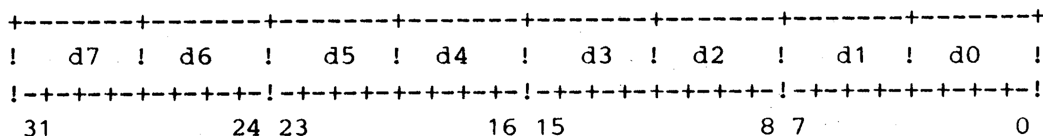
Packed Decimal instructions add and subtract packed decimal operands. The following is a list of the Packed Decimal instructions:

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

A packed decimal operand consists of 2, 4, or 8 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 modelled 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 [illegible] 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.