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 Format

The NS16000 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.

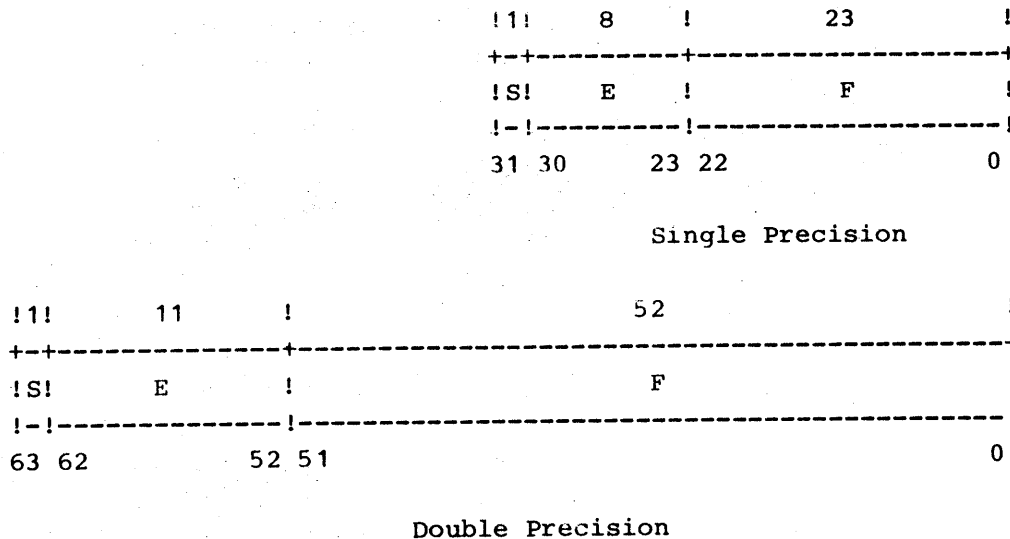


Figure 3-1 Floating-Point Operand Formats

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.

Tab[illegible] Ranges

	Single Precision	Double Precision
Most Positive	$2^{127} * (2 - 2^{-23})$	$2^{1023} * (2 - 2^{-52})$
	= 3.402823[illegible]	$= 1.797693134862316 * 10^{308}$
Least Positive	2^{-126}	2^{-1022}
	$= 1.1754943 * 10^{-38}$	$= [illegible].225073858507201 * 10^{-308}$
Least Negative	$-(2^{-126})$	$-(2^{-1022})$
	$= [illegible]1.[illegible]54943 * 10^{-38}$	$= -2.225073858507201 * 10^{-308}$
Most Negative	$-2^{127} * (2 - 2^{-23})$	$-2^{1023} * (2 - 2^{-52})$
	$= -3.4028235 * 10^{38}$	$= -1.797693134862316 * 10^{308}$

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. Despite the differing representations, 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 NS16000 hardware currently treats these forms as reserved operands. The reserved operands are:

1. Positive and Negative Infinity

2. Not-a-Number (NaN) values

3. 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 NS16000 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 NS16000 hardware does not generate reserved operands as results of floating-point calculations. The trapping mechanism used in the NS16000 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 [illegible]

Trap (UND)

The Floating-Point instruction set is made available to an NS16000-based system with an NS16081 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 condition 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 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 always sets the FSR 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 always sets the FSR IF bit unless any other error has occurred in the same instruction, in which case 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
<u>Arithmetic</u>		
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
<u>Shift</u>		
Arithmetic Shift	ASHB, ASHW, ASHD	ASHi
Logical Shift	LSHB, LSHW, LSHD	LSHi
Rotate	ROTB, ROTW, ROTD	ROTi
<u>Boolean</u>		
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 [illegible]gth 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)

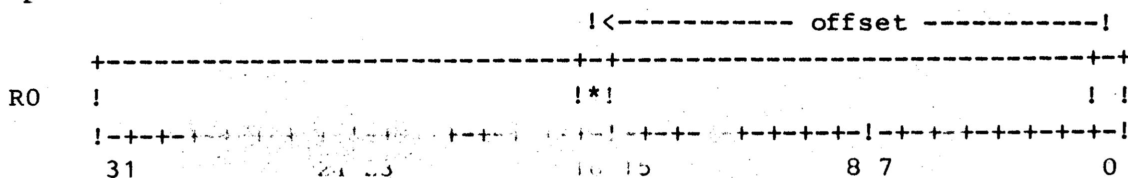
where "EA(base)" is the effective address calculated from the base operand specification and "offset DIV 8" is the nearest integer less than or equal to offset/8 (as per the DIVi instruction). The bit number of the desired bit is computed as

offset MOD 8

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

The following examples illustrate the interpretations of various bit specifications.

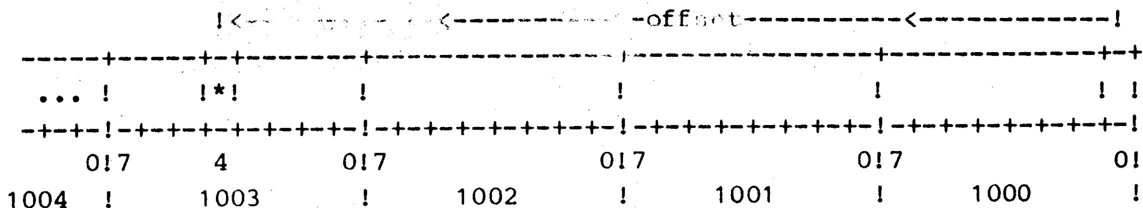
Example 1:



Offset : 16 Base : R0

Interpreted as bit 16 of register R0.

Example 2:

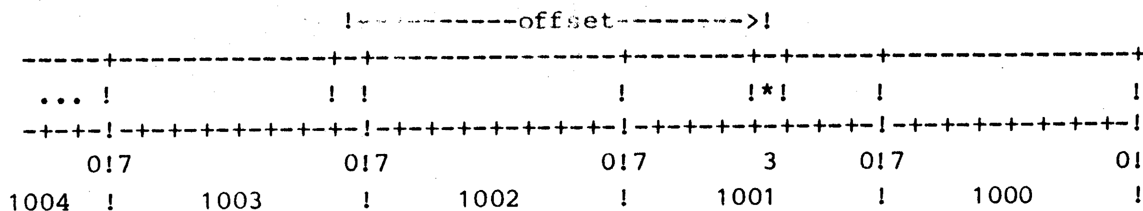


Offset: +28 EA(Base): 1000

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).

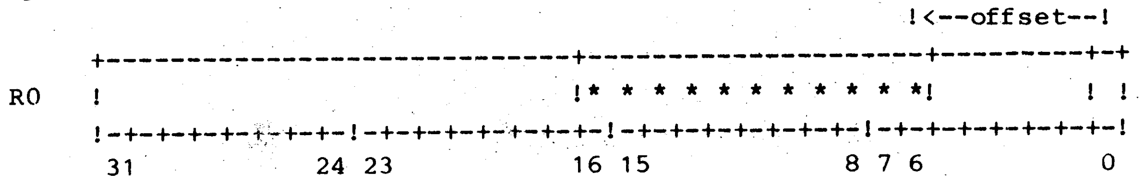
NOTES:

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, an Insert instruction always reads, modifies and rewrites a full double-word. This should be taken into consideration in multiprocessor systems where two processors may be accessing adjacent fields.

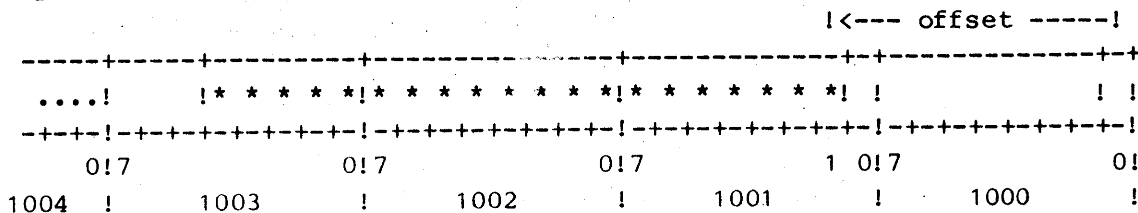
The following examples illustrate how a bit field is located in a register and in memory.

Example 1:



Base: R0 Offset: 6 Length: 11
Interpreted as an 11-bit field in register R0 starting with bit 6.

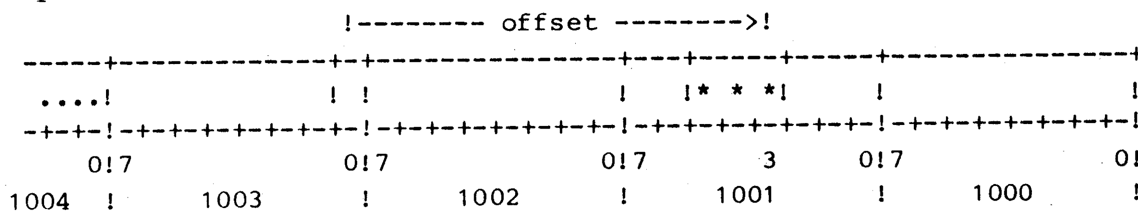
Example 2:



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 one-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 match 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:

```
+-------+---+---+
!  UW   ! B ! T !
+---+---+---+---+
```

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, or

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. The following is a list of the 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. See below.) Since a block must contain at least one byte and must contain 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 [illegible] 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 [illegible] 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 multi-dimensional arrays. Its purpose is to calculate a single one-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 [illegible] Major Ordering

Pascal array declaration:

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

Element size: 4 bytes
Base address: [illegible] (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

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 non-sequentially. 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 multi-way 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 NS16000 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 from 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 [illegible] and restoring of portions of the processor context to and from [illegible]. 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) [illegible] onto [illegible] current stack, allocates a specified number of bytes on the stack [illegible] 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.[illegible].) 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 one-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 I

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.

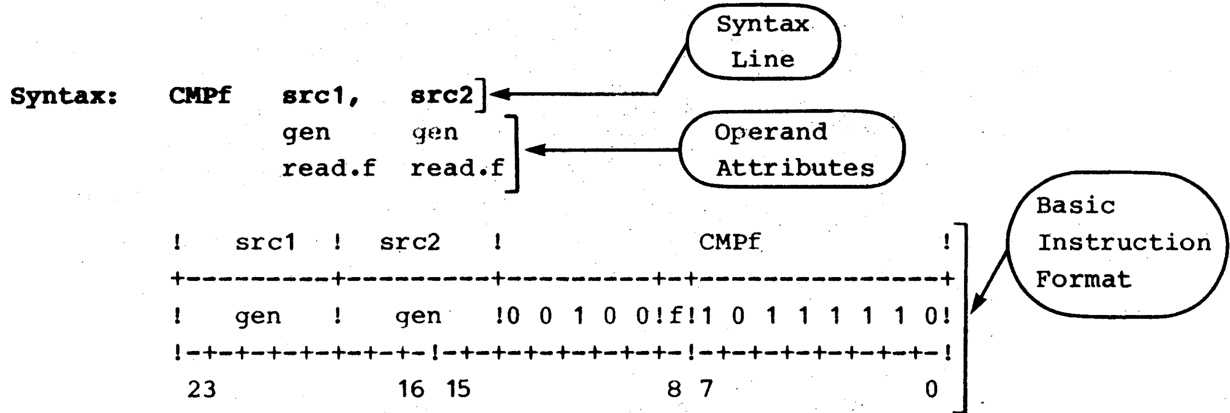
See the applicable CPU data sheet for details of the instruction formats and the Slave Processor protocols used.

Chapter 4

INSTRUCTION OPTIONS AND CONSTRUCTION

This chapter defines the options available in NS16000 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 is shown below:



The notations used are defined in the following sections:

Syntax Line 4.1

Operand Attributes 4.2

Instruction Format 4.3

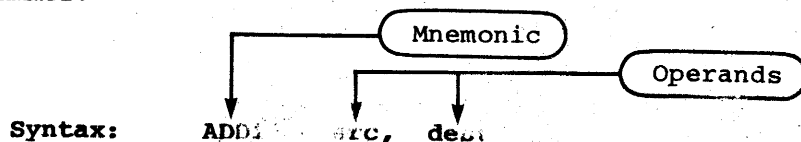
Other information presented in this chapter:

Addressing Modes 4.4

Construction Examples 4.5

4.1 Syntax Presentation

The Syntax line presents the instruction mnemonic, followed by a list of operands, as shown below. 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

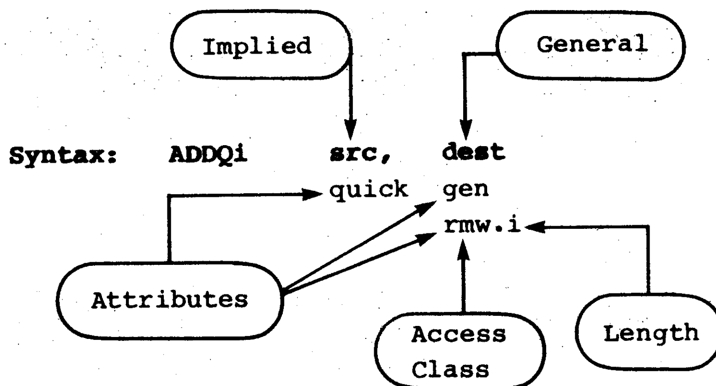
The specifications allowed and their interpretations are listed in the instruction description.

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 NS16000 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 NS16000 architecture defines five standard contexts (access classes) within which an NS16000 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	Push	Pop	(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 non-standard 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 one-byte integer.

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

D The operand is a four-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 four-byte single-precision floating-point value.

L The operand is an eight-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 one-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.

```
+---+---------------------------+
! 0 !     7-bit signed value    !      Range:  -64...+63
+---+---+---+---+---+---+---+---+

+---+---+-----------------------+
! 1   0 !      14-bit           !
+---+---+      signed       ----+      Range:  -8192...+8191
!              value            !
+---+---+---+---+---+---+---+---+

+---+---+-----------------------+
! 1   1 !                       !
+---+---+                   ----+      Range:  currently
!              30-bit           !      -16,277,215...+16,277,215.
+----          signed       ----+      Values outside this range
!              value            !      are currently undefined.
+----                       ----+
!                               !
+---+---+---+---+---+---+---+---+
```

4.3 Binary Instr[illegible]

The binary format of an NS16000 instruction is shown in Figure 4-1. It is divided into two sections.

1. The Basic Instruction portion, which 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 [illegible]

2. Extension fields, which 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 NS16000 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.