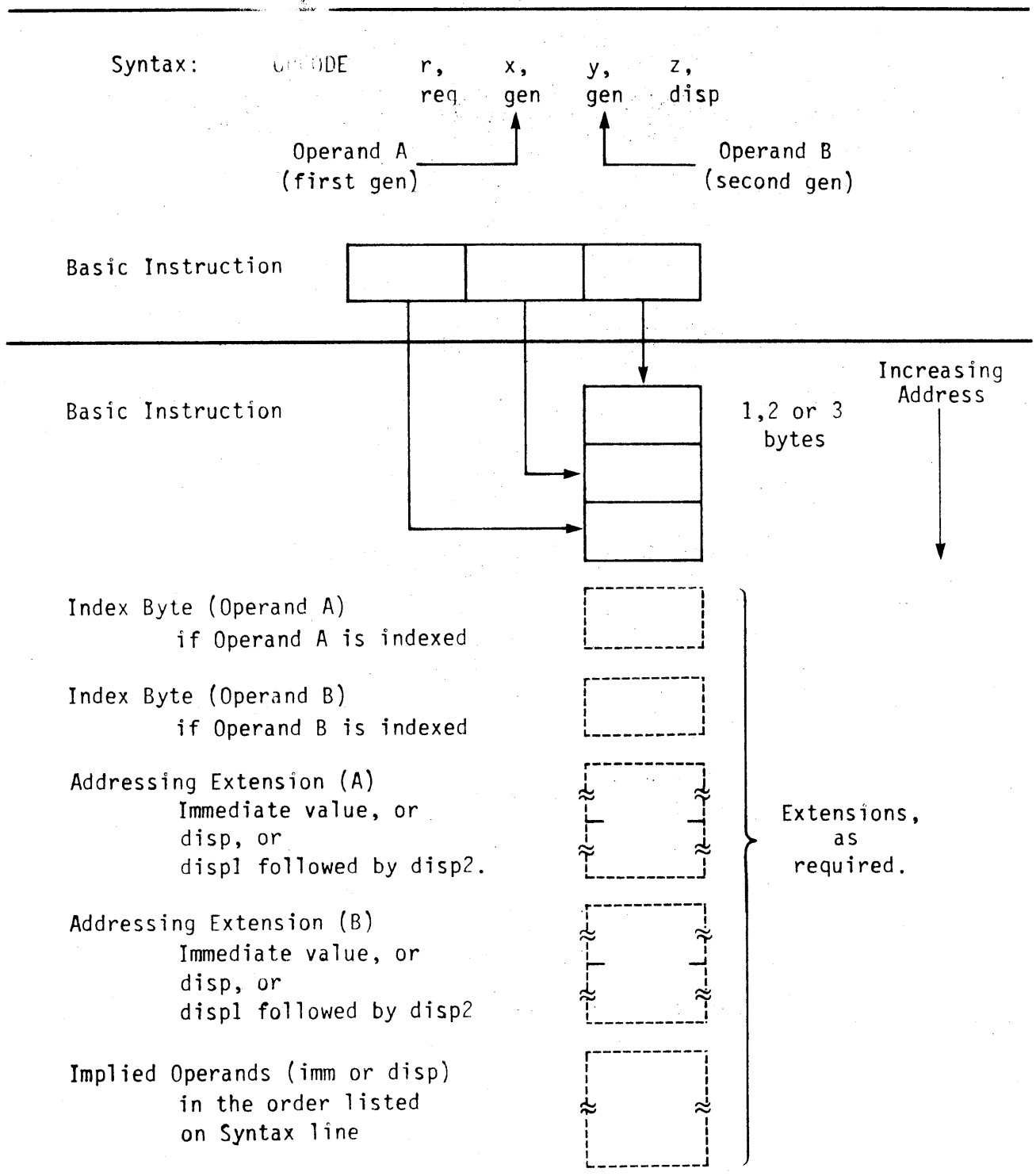


Figure 4-1 General Format

EZ-01-0

4.3.1 Basic Instruction

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

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

4.3.1.1 Operation Code Fields

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

4.3.1.2 Operation Length Fields: i and f

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

Field	Mnemonic Suffix	Encoding
i	B	00
	W	01
	D	11
f	[illegible]	1
	L	0

4.3.1.3 General Addressing Mode Fields: gen

These are 5-bit fields which define the addressing mode used to access each operand. The name of the operand from the Syntax line appears above the field. The encodings of these fields are given in the definitions of the addressing modes, Section 4.3.

4.3.1.4 Implied Operand Fields: reg, quick, short

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

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

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

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

4.3.2 Extension Fields

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

4.3.2.1 Index Bytes

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

4.3.2.2 Addressing Extensions

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

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

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

An addressing extension contains either:

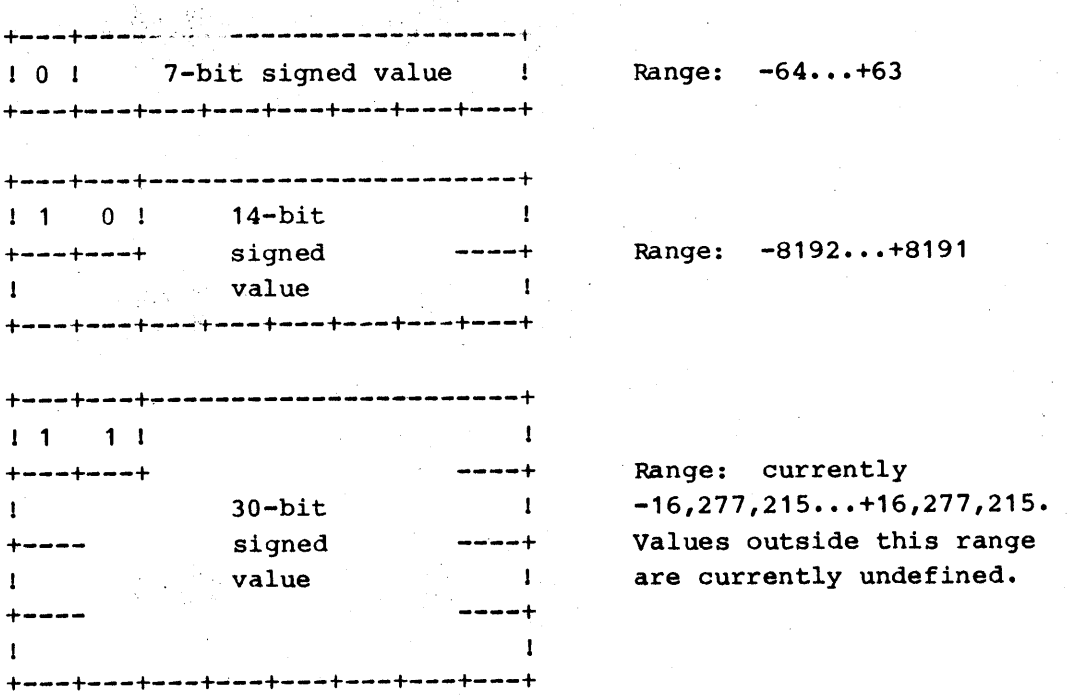
1. One immediate value, or

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

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

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

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



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

4.3.2.3 Implied Operand Extensions: imm, disp

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

4.4 NS16000 Addressing Modes

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

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

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

Addressing Modes

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

4.4.1 Register Mode

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

Extensions

None

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

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

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

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

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

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

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

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

4.4.2 Register Relative Modes

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

Extensions

One displacement field: disp.

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

4.4.3 Memory Rel[illegible]

Mode	Syntax	Encoding
Frame Memory Relative	disp2(disp1(FP))	10000
Stack Memo[illegible]elative	disp2(disp1(SP))	10001
Static M[illegible]tive	disp2(disp1(SB))	10010

Extensions

Two displacement fields: disp1 followed by disp2.

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

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

4.4.4 Immediate Mode

Mode	Syntax	Encoding
Immediate	value	10100

Extensions

The value, placed most-significant byte first.

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

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

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

4.4.5 Absolute Mode

Mode	Syntax	Encoding
Absolute	@address	10101

Extensions

One displacement field: address.

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

NOTE: Negative addresses are undefined.

4.4.6 External Mode

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

Extensions

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

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

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

4.4.7 Top of Stack Mode

Mode	Syntax	Encoding
Top of Stack	TOS	10111

Extensions

None

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

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

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

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

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

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

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

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

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

4.4.8. Memory Space Modes

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

Extensions

One displacement field: disp.

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

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

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

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

4.4.9 Scaled Inde

Mod	Syntax	Encoding
Byte Index 1	basemode[Rn:B]	11100
Word Indexed	basemode[Rn:W]	11101
Double-Word Indexed	basemode[Rn:D]	11110
Quad-Word Indexed	basemode[Rn:Q]	11111

basemode = base addressing mode (see below)

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

Extensions

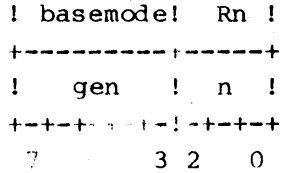
1. Index Byte.
2. Any extensions required by basemode.

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

basemode[Rn:l]

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

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



Any further addr[illegible]extensions required by basemode are appended as given in Section 4.3.2.2, in [illegible]tly the same manner as if basemode were not indexed.

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

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

4.5 Constructing [illegible] Some Examples

The following examples illustrate the process of assembling the binary form of an NS16000 instruction from its assembly-language form.

Example 1:

The simple example below is generated [illegible] the Move instruction (MOVi).

```
MOVB  R0, R1
```

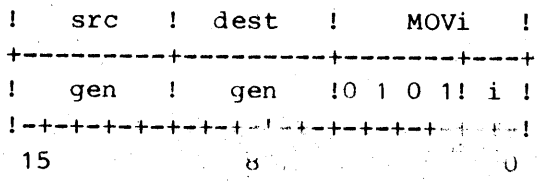
This instruction copies the low-order byte of register R0 to the low-order byte of register R1. The format definition of the MOVi instruction is taken from Chapter 5 as shown below.

Syntax: **MOVi** **src,** dest

gen gen

read.i write.i

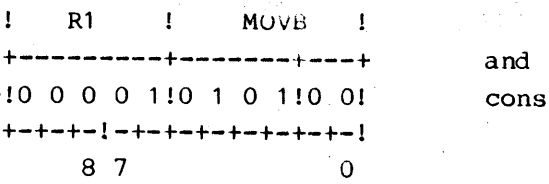
MOVB
MOVW
MOVD



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

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

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



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

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

Example 2:

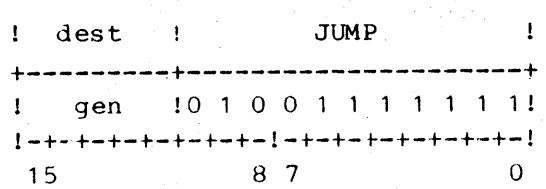
The next example is generated from the JUMP instruction.

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

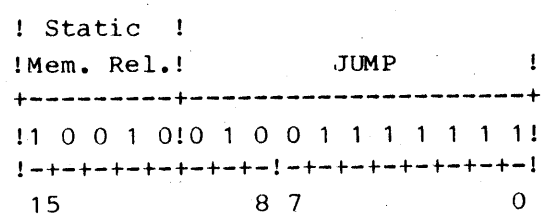
This instruction performs an indirect jump through a 32-bit pointer in memory. The pointer's address is calculated by adding 4 to the contents of the SB register.

The format definition of the JUMP instruction is:

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



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



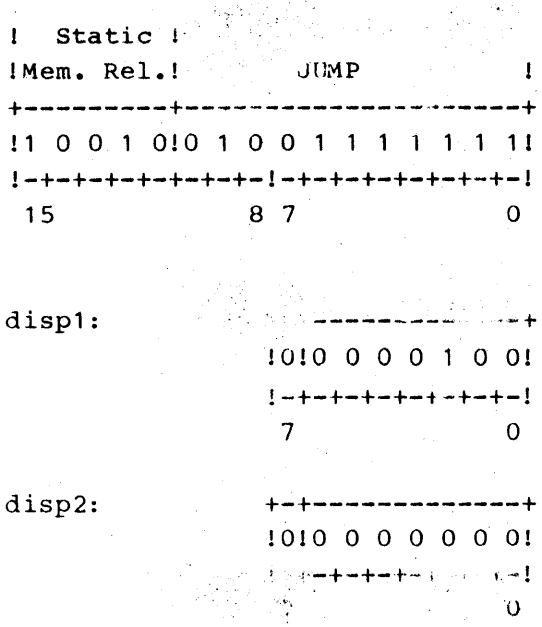
The Memory Relative addressing modes require that two displacements be appended to the Basic Instruction. These are designated disp1 and disp2. From the expression provided in the assembly-language example, the displacement values are to be:

disp1 = 4, and
disp2 = 0 .

(continued)

From the format q[illegible] tion 4.3.2.2), we see that a small value can be [illegible]ne, two or four bytes. Obviously, we wish to choose the smalle[illegible]hich works, so we will use the one-byte format for each displacement field.

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



The complete binary instruction is represented in consecutive memory bytes as

7F 92 04 00 (Hex).

Example 3:

The following example is generated from the ADDi instruction.

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

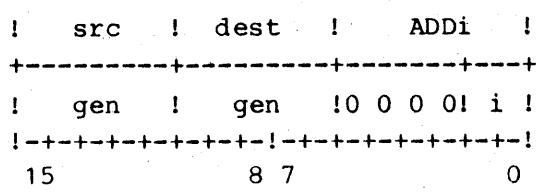
This instruction adds a 32-bit value from the memory location specified as EXT(8)+80 to a 32-bit value at the memory location specified as -4(FP).

The format definition of the ADDi instruction is:

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

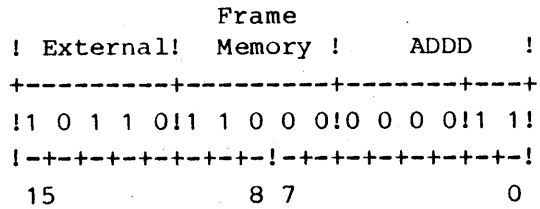
gen gen ADDW

read.i rmw.i ADDD



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

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



(continued)

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

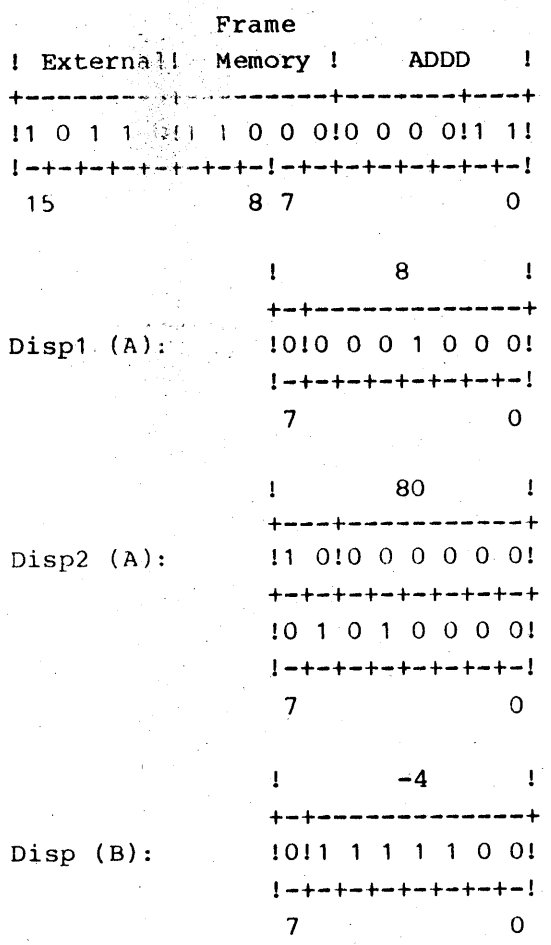
```
                 Frame
 ! External!  Memory !     ADDD    !
 +---------+---------+-------+---+
 !1 0 1 1 0!1 1 0 0 0!0 0 0 0!1 1!
 !-+-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
  15              8 7               0

                    !        8        !
                    +-+--------------+
 Disp1 (A):         !0!0 0 0 1 0 0 0!
                    !-+-+-+-+-+-+-+-!
                     7               0

                    !        80       !
                    +-+--------------+
 Disp2 (A):         !1 0!0 0 0 0 0 0!
                    +-+-+-+-+-+-+-+-+
                    !0 1 0 1 0 0 0 0!
                    !-+-+-+-+-+-+-+-!
                     7               0
```

(continued)

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



The complete instruction appears in consecutive memory bytes as:

03 B6 08 80 50 7C (Hex).

Example 4:

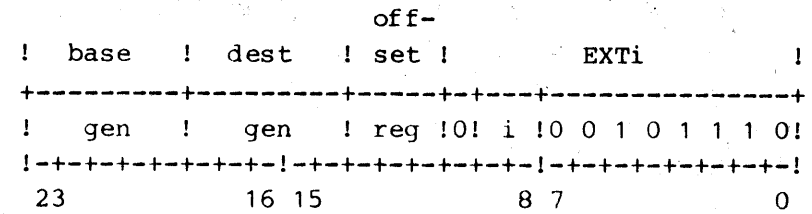
A final example of how an instruction is assembled uses the Extract Field (EXTi) instruction.

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

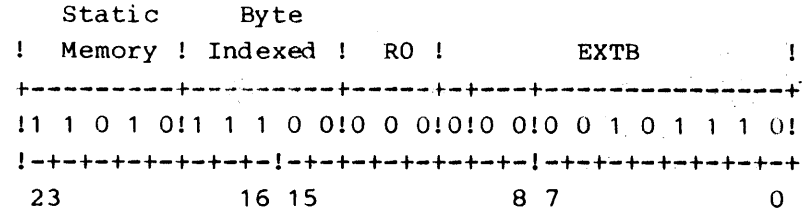
This instruction copies a 5 bit field from a point in memory determined by a bit offset (contained in R0) from the address 10(SB) to the address specified by 0(SB)[R1:B]. The format definition of the Basic Instruction is:

Syntax: **EXTi** **offset,** **base,** **dest,** **length**

	offset	base	dest	length
EXTB	reg	gen	gen	disp
EXTW		regaddr	write.i	
EXTD				

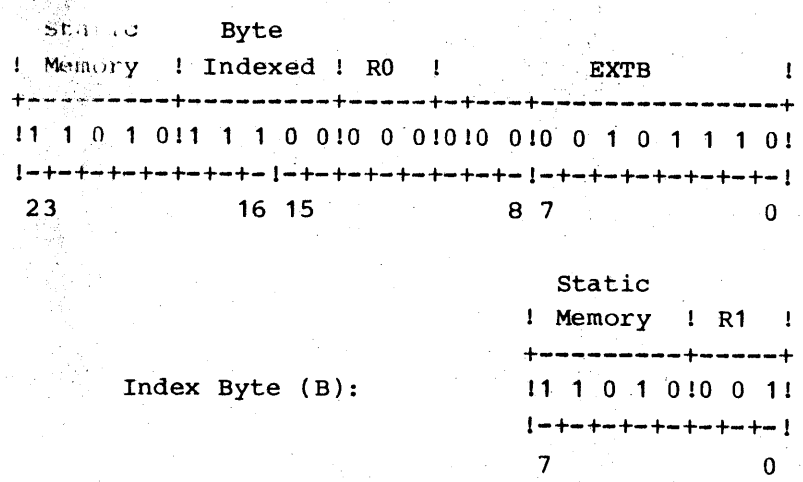


In this more complex instruction, there are several items which must be placed in the Basic Instruction. These are the addressing modes specified by the expressions 10(SB) and 0(SB)[R1:B], the i field corresponding to the B operation length suffix, and the reg field corresponding to the reg operand specified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



(continued)

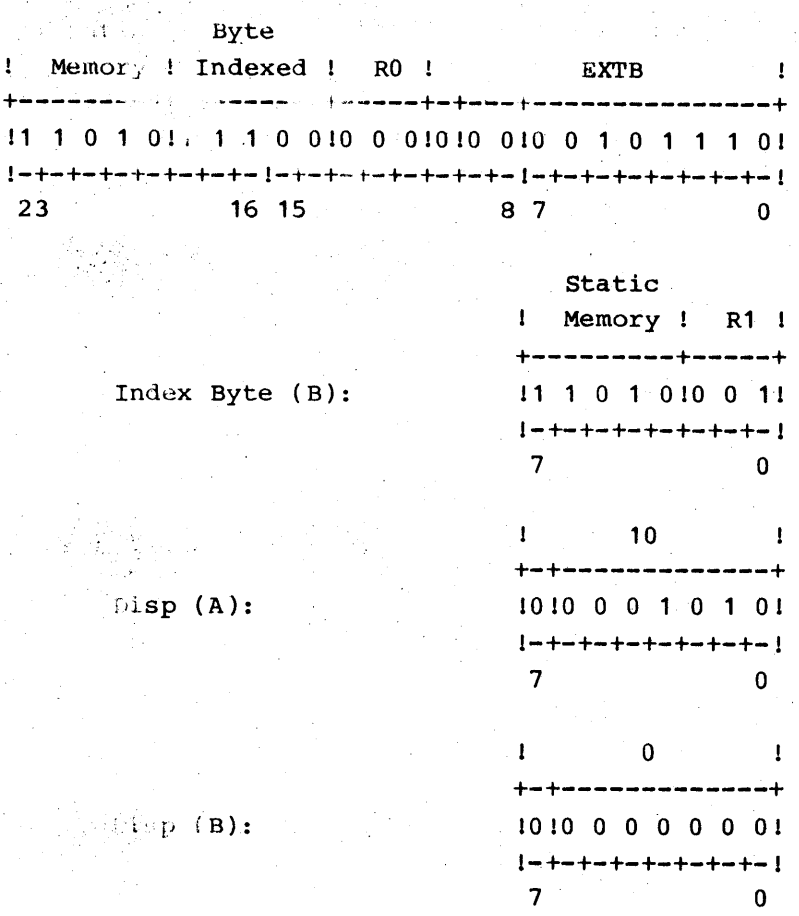
The expressi[illegible] specifies Operand A and 0(SB)[R1:B] specifies Operand B. Because it is [illegible] Operand B requires an Index Byte. The Index Byte is the first extensio[illegible] appended to the Basic Instruction. It contains the code for the basemo[illegible] and the register number for R1. The basemode (Static Memory) is encod[illegible] 1010 and the register number is encoded for R1 as 001.



(continued)

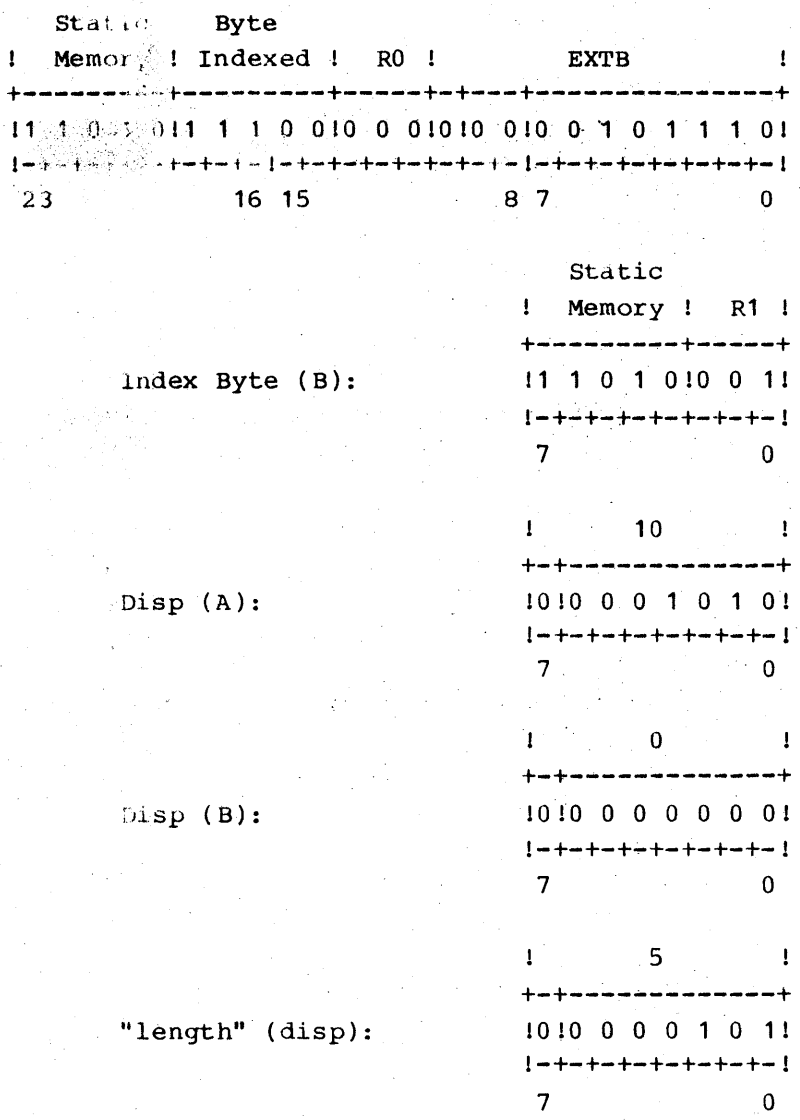
The next extension [illegible] the addressing extensions required by the addressing mode [illegible] general operands. Since Operand A is specified using the Static Memory addressing mode, it requires one displacement field, containing 10. This displacement is placed in single-byte format after the Index Byte.

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



(continued)

Finally, the le[illegible] fixed as 5) is an implied displacement which is appended after [illegible] extensions. It also can be encoded in single-byte format due to [illegible]tents. Thus, the complete machine instruction is:



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

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

Chapter 5

NS16000 INSTRUCTION SET

This chapter contains the detailed definitions of each of the instructions in the NS16000 instruction set.

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

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

2. Enumerated mnemonics. When there are multiple forms, this area holds a list of all valid mnemonic forms for the instruction.

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

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

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

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

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

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

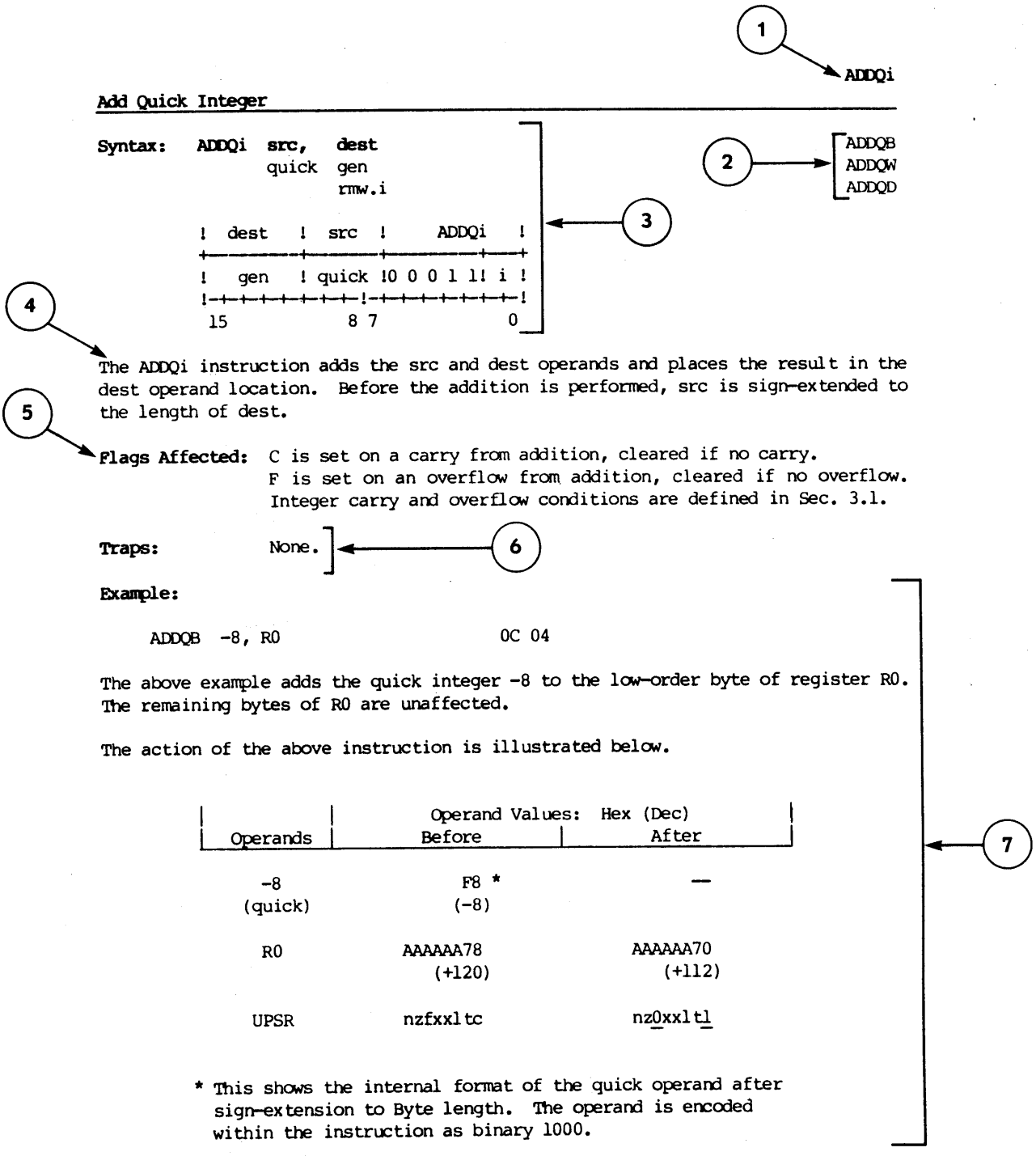


Figure 5-1: Typical Instruction Definition

5.1 Instruction Examples

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

5.1.1 Coding Examples

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

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

5.1.2 Action Examples

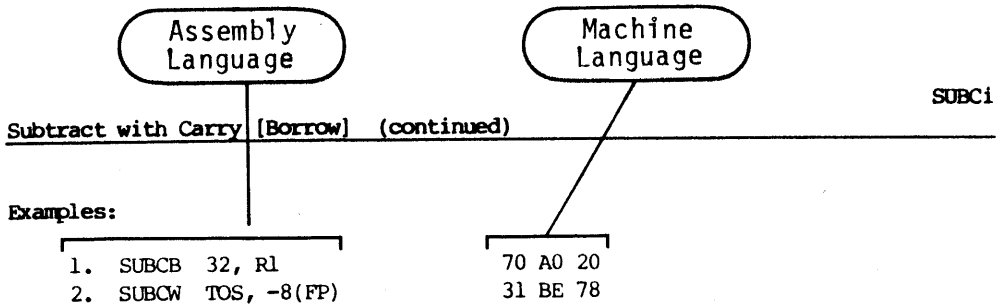
The actions of an example instruction are shown in three columns.

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

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

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

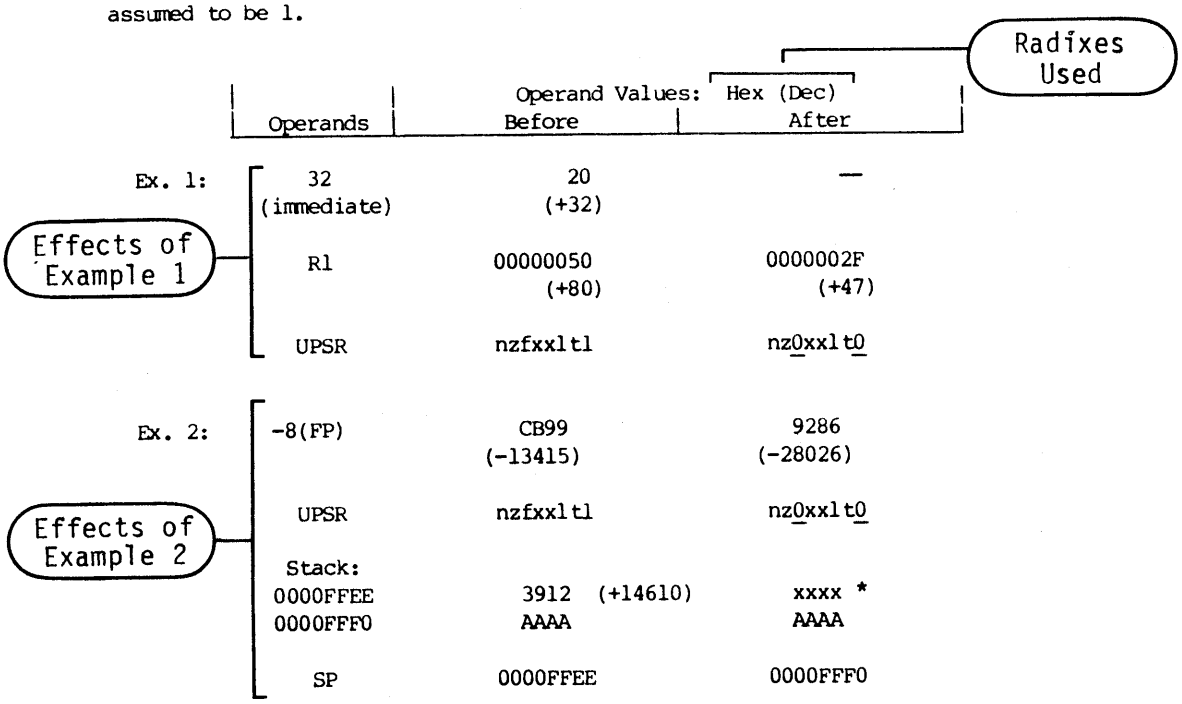
NOTE: An immediate or displacement value is not considered to have an "After" value, even though it never changes, because it is not available as an immediate or displacement value to any subsequent instructions.



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

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

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



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

Figure 5-2: Typical Instruction Example

5.1.3 Operand Presentation Format

The memory format convention used by the NS16000 family places the least-significant byte of a memory operand at the first (i.e. lowest) address. The correct interpretation of a multiple-byte value in memory, therefore, is produced by assembling consecutive bytes of the value from right to left. The address of an operand in memory is also the address of its least-significant byte.

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

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

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

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

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

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

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

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

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

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

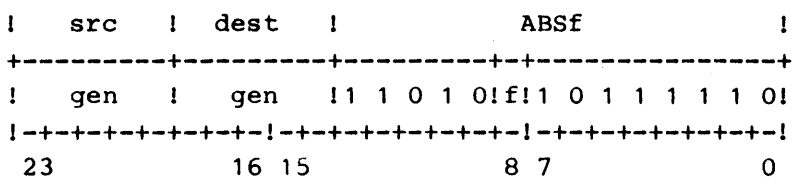
5.2 Instruction Definitions

This section defines the individual NS16000 instructions. The instructions are ordered alphabetically by their general mnemonic form. For listings of instructions by functional groups, see Appendix A. For help in interpreting the information presented here, see the beginning of this chapter.

ABSf

Absolute Value Floating

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



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

Flags Affected: None.

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

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

Example:

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

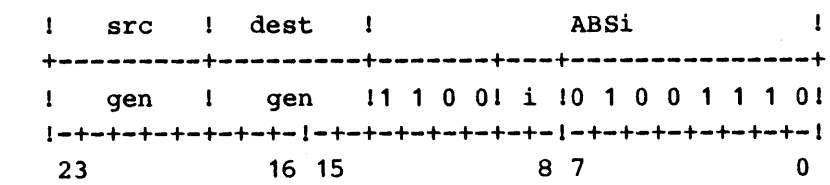
This example computes the absolute value of the single-precision number in register F0 and places the result in register F2.

The instruction is illustrated below:

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

Absolute Value

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



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

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

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

C is not affected.

Traps: None.

Examples:

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

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

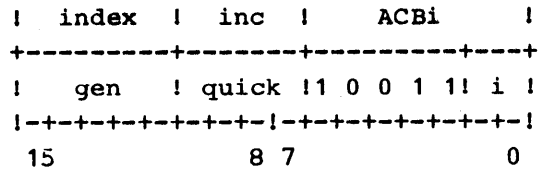
Example 2 computes the absolute value of the double-word at the memory address specified by 8(SP) and places the result in register R7.

These instructions are illustrated below:

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

Add, Compare and Branch

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



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

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

Flags Affected: None.

Traps: None.

Example:

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

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

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

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

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

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

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