Section 3

INSTRUCTION SYNTAX

3.1 INTRODUCTION

3.1.1 General

For assembly purposes the instruction repertoire of the ALPHA series computers are divided into nine classes. These classes are generally the same as those described in the NAKED MINI LSI/ALPHA LSI Programming Reference Manual. Refer to that document for functional descriptions of each instruction and of memory addressing modes.

3.1.2 Instruction Classes

The instruction classes are discussed in this section in a logical sequence rather than in their numerical sequence. The instruction classes and their sequence of discussion are as follows:

Class 1 - Memory Reference, Word Operand
Class 8 - Memory Reference, Byte Operand
Class 9 - Arithmetic, Double Word
Class 2 - Immediate
Class 3 - Conditional Jump
Class 4 - Shift, Single Register
Class 7 - Long Shift
Class 5 - Register Change and Control
Class 6 - Input/Output

3.1.3 Syntax Description

This section describes the syntax for each instruction class. In the following descriptions, brackets are used to indicate optional fields.

3.2 CLASS 1 - MEMORY REFERENCE, WORD OPERAND

3.2.1 General

Class 1 instructions process full-word operands. Instruction descriptions and addressing modes are described in section 3.2 of the NAKED MINI LSI/ALPHA LSI Programming Reference Manual.

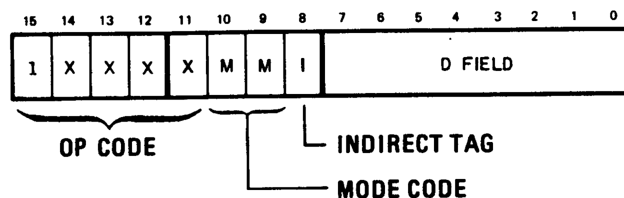


Figure 3-1. Class 1 Machine Language Format

3.2.2 Assembly Format

Class 1 instruction format for assembly purposes is as follows:

[LABEL] OP CODE OPERAND [COMMENTS]

3.2.2.1 Label Field

The Label field is optional with Class 1 instructions.

3.2.2.2 Operation Code Field

The Operation Code (Op Code) field must be present. Legal Op Codes for Class 1 instructions are:

ADD	Add to A
AND	Logical AND with A
CMS	Compare memory and A
EMA	Exchange memory and A
IMS	Increment memory and Skip
IOR	Inclusive OR with A
JMP	Jump
JST	Jump and Store
LDA	Load A
LDX	Load X
STA	Store A
STX	Store X
SUB	Subtract from A
XOR	Exclusive OR with A

3.2.2.3 Operand Field

The operand field must be present. With the exception of literals, the expression in the operand field represents a memory word address. The expression may be preceded by an asterisk (*) to denote indirect addressing, an "at" sign (@) to denote indexing, or an equal sign (=) to denote a literal.

3.2.2.4 Comment Field

The comments field is optional.

3.2.2.5 Examples

The following are examples of Class 1 instructions.

a. Example 1. All fields used:

```
A1      LDA        ABC   This is a comment.
```

b. Example 2. Indirect addressing specified:

```
        LDA   *ABC
```

c. Example 3. Indexing specified:

```
        LDA   @0
```

d. Example 4. Literal operand:

```
        LDA   =:F6
```

e. Example 5. External:

```
        EXTR  A
        LDA   A
        LDA   *B
B       REF
```

3.2.3 Memory Addressing Modes

Figure 3-1 illustrates the machine language format for Class 1 instructions. The assembler evaluates the expression in the operand field of a Class 1 instruction statement and uses that information, along with the current contents of the location counter, to determine the contents of the D, I, and M fields of the assembled instruction.

3.2.3.1 Indirect (I) Field

The indirect flag, I bit, is set to 1 if either or both of the following conditions exists:

a. An asterisk (*) proceeds the expression in the operand field of the source statement.

b. The expression value is not within direct addressing range.

In the first case, the indirect flag is unconditionally set, and the expression is evaluated to determine the values of the M and D fields. If the expression value is within direct addressing range, the M and D fields are assembled for scratchpad or relative to P addressing to address the indirect address pointer (see paragraph 3.2.3.2 and 3.2.3.3 for scratchpad and relative to P addressing).

If the expression value is not within direct addressing range, the assembler sets the indirect bit on and the object loader, at load time, will generate an address to a scratchpad location which will be loaded with the out-of-range address. For example:

```
:150        LDA    ABC
  .          .
  .          .
  .          .
  .          .
  .          .
:3AB  ABC   DATA   0
```

The symbol ABC has the value :3AB, which is not within direct addressing range of the instruction at location :150. The assembled machine language for the instruction is:

```
:B100       (The instruction)
:03AB       (The out-of-range address)
```

At load time the object loader will place the out-of-range address in scratchpad, and will place the address of the word containing the out-of-range address in the D field of the instruction.

If the programmer specifies indirect addressing and the expression value is out-of-range, multi-level indirect addressing is required. For example:

```
:150        LDA    *ABC
  .          .
  .          .
  .          .
:3AB  ABC   DATA   0'
```

The symbol ABC has the value :3AB which is not within direct addressing range of the instruction at :150. The assembler must generate an indirect address pointer to address the word at :3AB, but the word at :3AB also contains an indirect address pointer. Therefore, multi-level indirect addressing is required.

The assembled machine language for the instruction at location :150 is:

:B100	(The instruction)
:83AB	(The out-of-range address with the multi-level indirect bit on).

At load time the object loader will place the out-of-range address in scratchpad and will place the address of the word containing the out-of-range address in the D field of the instruction.

3.2.3.2 Scratchpad Addressing

If the expression value is within the range, 0 to 255 (:00 to :FF), the memory location may be addressed directly using scratchpad addressing. For example:

```
LDA        :A5
```

In the above statement the expression value is within the scratchpad area of memory. The assembled machine language code for this instruction is:

```
:B0A5
```

The statement appears as follows if indirect addressing is specified:

```
LDA        *:A5
```

The assembled machine language code is:

```
:B1A5
```

3.2.3.3 Relative Addressing

Relative addressing is generated if the expression value is within the range -255 to +256 locations from the current value of the location counter. Relative to P forward addressing is generated if the expression value is in the range:

(Location Counter) + 1 thru (Location Counter) + 256

Relative to P backward addressing is generated if the expression value is in the range:

(Location Counter) - 255 thru (Location Counter)

The following program sequence causes relative to P backward addressing:

```
:100    ABC    DATA 0
:101              .
:102              .
:103              .
:104           LDA ABC
```

The symbol ABC has the value :100, which is not in scratchpad. When the instruction at location :104 is assembled, the location counter value is :104 and the expression value is :100. Subtracting the location counter value from the expression value gives:

:100 - :104 = - :04

This value is within range for relative P backward addressing. The machine language for the instructions at location :104 is:

:B604

The following program sequence causes relative to P forward addressing:

```
:100           LDA  DEF
:101             .
:102             .
:103             .
:104    DEF    DATA 0
```

The symbol DEF has the value :104, which is not in scratchpad. When the instruction at location :100 is assembled, the location counter value is :100 and the expression value is :104. Subtracting the location counter value from the expression value gives:

:104 - :100 = :04

This value is within range for relative to P forward addressing. Relative to P forward addressing is defined as:

(P) + 1 + (D)

Therefore, a 1 must be subtracted from the above value to obtain the correct value to be placed in the D field of the assembled instruction:

: 04 - 1 = : 03

The machine language for the instruction at location : 100 is:

: B203

If the expression in the above example had been preceded by an asterisk (LDA *DEF), the assembled machine language would be:

: B303

3.2.3.4 Indexing

Indexing is specified if the expression is preceded by an "at" sign (@). If indirect addressing is not specified, two possible conditions exist:

a. The expression value may be in the range 0 to 255 (: 00 - : FF).

b. The expression value may be 256 (: 100) or greater. In this case the assembler generates information to the loader which will cause an indirect address pointer to be generated at load time. The assembler will provide the indirect address pointer value, and the loader will provide linkage at load time.

If indexed and indirect addressing is specified, the same two conditions effectively exist:

a. If the expression value is in the range 0 to 255, only one level of indirect addressing is required.

The following example contains an expression in the range 0 to 255:

```
LDA    @5
```

The assembled machine language for the instruction is:

: B405

The following example contains an expression in the range 0 to 255 with indirect addressing specified:

```
LDA    @*5
```

The assembled machine language for the instruction is:

:B505

The following program sequence generates an expression which is 256 or greater:

```
:150          LDA   @TBL
:151            .
:152            .
:153            .
:154    TBL   RES   4
:155            .
  .             .
```

The symbol TBL has the value :154. Since this value is greater than 255 (:FF), the assembler must provide indirect addressing information for the Object Language Loader.

The assembled machine language for the instruction at location :150 is:

:B500 (Instruction)
:0154 (Out-of-range address)

At load time the object loader will place the out-of-range address in a location in scratchpad, and will place the address of the scratchpad word in the D field of the instruction.

The following program sequence generates multi-level indirect addressing:

```
:2A0          LDA   *@ADR
:2A1            .
:2A2            .
:2A3            .
:2A4    ADR   DATA TBL
```

The symbol ADR has the value :2A4 which is greater than :FF. The assembled object code is the same as for single level indirect addressing, except that the multi-level indirect address bit must be set in the assembler - generated indirect address pointer:

address pointer = :2A4 + :8000 = :82A4

The assembled machine language for the instruction at location :2A0 is:

:B500 (The instruction)
:82A4 (The out-of-range address with the multi-level indirect bit on.)

At load time the object loader will place the out-of-range address in a word in scratchpad and will place the address of that word in the D field of the instruction.

3.3 CLASS 8 - MEMORY REFERENCE, BYTE OPERAND

3.3.1 General

Class 8 instructions process byte operands. Byte Mode memory reference instructions and byte operand addressing are described in section 3.2 of the NAKED MINI LSI/ ALPHA LSI Programming Reference Manual.

3.3.2 Assembly Format

Class 8 instruction format for assembly purposes is as follows:

[LABEL] OP CODE OPERAND [COMMENTS]

3.3.2.1 Label Field

The Label field is optional with Class 8 instructions.

3.3.2.2 Operation Code Field

Class 8 instructions perform the same functions as Class 1 instructions except that they process byte operands instead of word operands. Also, memory addressing modes are different. Byte operand addressing modes are specified by adding the letter "B" to some of the memory reference op codes. Class 8 op codes which are recognized by the assembler are as follows:

ADDB	Add byte to A
ANDB	AND byte with A
CMSB	Compare bytes, A and memory
EMAB	Exchange memory and A, bytes
IORB	Inclusive OR byte to A
LDAB	Load byte to A
LDXB	Load byte to X
STAB	Store byte from A
STXB	Store byte from X
SUBB	Subtract byte from A
XORB	Exclusive OR byte to A

Word Address	Byte 0	Byte 1
ABC	2(ABC)	2(ABC)+1
	2(ABC)+2	2(ABC)+3
	2(ABC)+4	2(ABC)+5

Figure 3-2. Byte Addresses

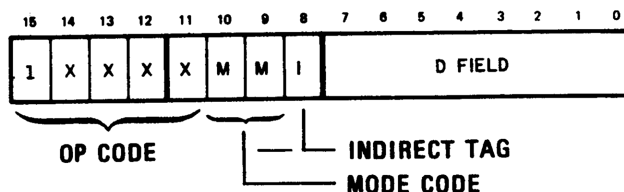


Figure 3-3. Class 8 Machine Language Format

3.3.2.3 Operand Field

The operand field must be present. The operand field of Class 8 instructions represents a byte operand address. Indirect addressing and indexing may be used with Class 8 operand field expressions. Literals may not be used.

Expressions in Class 8 operand fields are evaluated differently from Class 1 operand fields, since the address generated for Class 8 instructions represents a byte operand address rather than a word operand address. The general rules for expression evaluation are as follows:

a. Symbolic terms represent word locations. The values assigned to symbolic terms are doubled to form a corresponding byte address.

b. Numeric terms represent byte addresses. These values are not doubled when evaluating an expression.

Figure 3-2 illustrates byte locations within words. If the symbol ABC is the address of a word in memory, then 2(ABC) is the address of byte 0 of that word and 2(ABC) +1 is the address of byte 1 of that word. 2(ABC) + 2 is the address of byte 0 of the next word, and so on.

The assembler automatically performs the function of doubling the symbolic addresses which are contained in an expression. Therefore, to address byte 0 (the left byte) of the word at ABC, the programmer writes:

```
LDAB     ABC
```

To address byte 1 of the word at ABC, the programmer writes:

```
LDAB     ABC+1
```

In both cases the assembler doubles the value of ABC since ABC is a symbolic term. In the first case the assembler evaluates the expression as 2(ABC), and in the second case as 2(ABC) + 1.

3.3.2.4 Comments Field

The comments field is optional for Class 8 instructions.

3.3.3 Memory Addressing Modes

Figure 3-2 illustrates the machine language format for Class 8 instructions.

Relative to P backward addressing is not possible with Class 8 instructions. Relative to P forward has a range of from 1 to 512 bytes. The M field specifies specific byte 0 or byte 1 for relative to P forward addressing.

3.3.3.1 Indirect Addressing

Only one level of indirect addressing is possible with Class 8 instructions. Indirect addressing will be generated if either of the following conditions exists:

a. An asterisk (*) precedes the expression in the operand field.

b. The expression value is not within range for direct addressing.

The programmer must be careful that these two conditions do not exist simultaneously, since each condition would require a level of indirect addressing, and multi-level indirect addressing is not possible with Class 8 instructions.

For indirect addressing, the indirect address pointer is a full 16-bit word which contains a byte address. When the programmer specifies indirect addressing, the expression value must be treated as a word address rather than a byte address by the assembler. The reason is that the expression value represents the address of an indirect address pointer rather than a byte operand. For example:

```
:101              LDAB    *TBL
:102              ADDB    *TBL + 1
:103                .
:104                .
:105      TBL     BAC     BYTEA, BYTEB
:106
```

The instruction located at location :101 (LDAB *TBL) references location :105. The instruction at :102 references location :106. Locations :105 and :106 contain byte addresses.

In the following example indirect addressing is not specified by the programmer, but the byte address is not within direct addressing range of the instruction:

```
:200      TAG     DATA      0
:201              LDAB      TAG+1
```

The symbol TAG has the value :200. The expression in the operand field of the statement at location :201 is evaluated as follows:

2(TAG) + 1 = 2(:200) + 1 = :401

This value is not within range for direct addressing, so the assembler generates an indirect address pointer which contains the expression value. The assembled instruction machine language is as follows:

```
:B100     (The instruction)
:0401     (The out-of-range address)
```

At load time the object loader will place the out-of-range address in a word in scratchpad, and will place the address of that word in the D field of the instruction.

3.3.3.2 Scratchpad Addressing

If indirect addressing is not specified, and if the expression value is in the range 0 to 255 (: 00 to : FF), then the byte operand may be addressed directly using scratchpad addressing.

The expression value of the following statement falls in this category:

```
            LDAB      :A3
```

The assembled object code is:

```
        :B0A3
```

If indirect addressing is specified, the indirect address pointer may be in scratchpad. For example:

```
            LDAB      *:EA
```

The complete machine language instruction is:

```
        :B1EA
```

Note: The D field contains a word address.

3.3.3.3 Direct Relative Addressing

Bytes may be addressed directly relative to a Class 8 instruction if the byte location is from 1 to 512 bytes forward from the instruction location; i.e., if the byte address is in the range:

2(Location Counter) + 1 thru 2(Location Counter) + 512

The following example illustrates relative to P addressing:

```
:201                  LDAB    BYTEA
:202                  ADDB    BYTEA+1
:203                    .
:204        BYTEA     DATA    :0102
```

The symbol BYTEA has the value : 204. The expression in the statement at location : 201 is first evaluated as:

2(BYTEA) = 2(: 204) = : 408

The least significant bit (LSB) of the byte address (: 408) is a 0, therefore the left byte (byte 0) of a word is being addressed. The address of the word which contains the byte being addressed is:

(Byte address)/2 = (: 408)/2 = : 204

Relative to P byte addressing is possible if the <u>word</u> address is within the range of 1 thru 256 (: 01 thru : 100) words forward from the location counter value. In this case:

: 204 - : 201 = 3

Relative to P addressing is defined as:

(P) + 1 + (D)

The assembled machine language is:

: B202

The expression in the statement of : 202 is evaluated as follows:

2(BYTEA) + 1 = 2(: 204) + 1 = : 408 + 1 = : 409

The byte address is connected to a word address for relative addressing:

(: 409)/2 = : 204, remainder of 1

The LSB of the byte address is 1(the remainder after the division), therefore the right byte (byte 1) of the word at location : 204 is being addressed. The relative distance is computed as:

: 204 - : 202 = 2

2-1 = 1

The assembled machine language is:

: 8E01

3.3.3.4 Indirect Relative Addressing

An indirect address pointer for indirect byte addressing may be located relative to P forward or backward. The range is -255 thru +256 words from the location counter value.

The assembly of an instruction in this case is identical to a Class 1 instruction except for the evaluation of the expression in the operand field. The expression is evaluated the same as for any other Class 8 instruction. For example:

```
:301           LDAB   *TAG
:302             .
:303    TAG    DATA   0
:304           DATA   0
:305           LDAB   *TAG+1
:306
```

The expression in the statement at location :301 is evaluated as follows:

2(TAG)=2(:303)=:606

(:606)/2 = :303

The indirect address pointer is within range for relative forward addressing:

:303 - :301 = 2

2-1=1 (Since the distance is relative to P + 1)

The assembled machine language is:

:B301

The expression in the statement at location :305 is evaluated as follows:

2(TAG) + 2= 2(:303) + 2=:606 + 2 = :608

(:608)/2 = :304

This is the address of the word which follows TAG. The address is within range for relative to P backward addressing:

:304 - :305 = -1

The assembled machine language is:

:B701

3.3.3.5 Indexing

When indexing is specified with a Class 8 instruction, the assembly is performed the same as for a Class 1 instruction except that the operand field expression is evaluated as a byte address. For example:

```
:400                LDAB    @:5A
:401                LDAB    @TBL
:402                LDAB    @TBL+1
:403                 .
:404       TBL      DATA    :ABAC
```

The expression in the statement at location :400 is absolute. The assembled machine language is:

:B45A

The instruction at location :401 has a symbolic expression which must be evaluated. The symbol TBL has the value :404, therefore:

2(TBL) = 2(:404) = :808

This value is too large for the D field, so an indirect address pointer must be generated:

Address Pointer = :0808 (The expression value)

The assembled machine language is:

```
:B500       (The instruction)
:0808       (Out-of-range address)
```

At load time the object loader will place the out-of-range address pointer in a word in scratchpad, and will place the address of that word in the D field of the instruction.

The instruction at location :403 is assembled in exactly the same manner as the one at location :402, except that the expression is evaluated as follows:

2(TBL) + 1 = 2(:404) + 1 = :808 + 1 = :809

The assembled machine language is:

```
:B500
:0809
```

If indexing is specified along with indirect addressing, then the indirect address pointer must be in scratchpad, since multi-level indirect addressing is invalid:

```
LDAB        *@:FF
```

The assembled machine language is:

:BF55

The following is an example of an instruction statement which cannot be assembled:

```
LDAB        *@:200
```

The address pointer is not in scratchpad. A second level of indirect addressing would be required to address it, and multi-level indirect addressing is not possible with Class 8 instructions. Therefore, this instruction cannot be assembled.

3.4 CLASS 9: MEMORY REFERENCE, DOUBLE WORD

3.4.1 General

Class 9 instructions require two consecutive words of memory and allow direct and indirect addressing only. Indexed addressing is not useful since these instructions manipulate both the A and X registers. Double word memory reference instructions and addressing modes are described in section 3.3 of the NAKED MINI LSI/ALPHA LSI Programming Reference Manual.

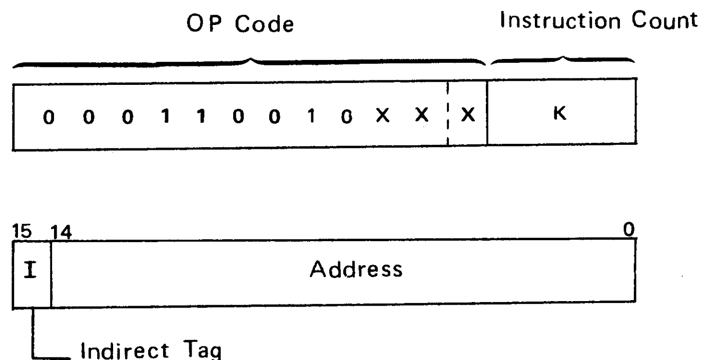


Figure 3-4. Class 9 Machine Language Format

3.4.2 Assembly Format

Class 9 instruction format for assembly purposes is as follows:

[LABEL] OP CODE OPERAND [, COUNT] [COMMENTS]

3.4.2.1 Label Field

The label field is optional.

3.4.2.2 Operation Code Field

The operation code (OP Code) field must be present. Legal OP Coder for Class 9 instructions are:

DVD	Divide
MPY	Multiply and Add
NRM	Normandize A and X

3.4.2.3 Operand Field

The operand field consists of one or two expressions, the first of which must be present. The first expression represents a memory word address, and may be preceded by an asterisk (*) to denote indirect addressing. This is the only addressing mode allowed.

The second expression is optional and must be separated from the first by a comma when used. This expression represents an optional instruction count in the range 1 through 16 for DVD and MPY and 0 through 31 for NRM. Omission of the expression will cause generation of a full multiply, divide or normalize instruction.

3.4.2.4 Comment Field

The comments field is optional.

3.4.2.5 Class 9 Examples

The following are examples of Class 9 instructions.

a. Example 1. A "normal" multiply and add:

```
LABEL    MPY    BASE     MULTIPLY X BY BASE, ADD A
```

b. Example 2. Indirect addressing specified:

```
LI       DIV    *PTR     DIVIDE A AND X BY VALUE
                         POINTED TO BY PTR
```

c. Example 3. Optional count specified:

```
L2       NRM    EXP,0    TEST A AND X FOR
                         NORMALIZED
```

3.5 CLASS 2 - IMMEDIATE INSTRUCTIONS

3.5.1 General

Class 2 instructions are similar to Class 1 and Class 8 instructions, except that the D field is interpreted as data rather than memory address information. Refer to subsection 3.4 of the NAKED MINI LSI/ALPHA LSI Programming Reference Manual for a full description of the Class 2 immediate instructions.

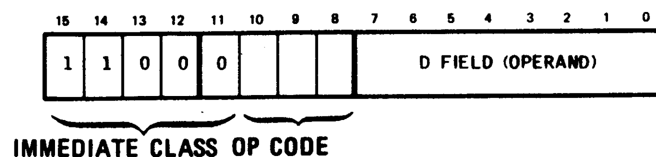


Figure 3-5. Class 2 Machine Language Format

3.5.2 Assembly Format

Class 2 source code format recognized by the assembler is as follows:

[LABEL] OP CODE OPERAND [COMMENTS]

3.5.2.1 Label Field

The Label field is optional with Class 2 instructions.

3.5.2.2 Operation Code

The operation code field must contain one of the following Class 2 operation codes:

AAI	Add to A Immediate
AXI	Add to X Immediate
CAI	Compare to A Immediate
CXI	Compare to X Immediate
LAM	Load A Minus Immediate
LAP	Load A Positive Immediate
LXM	Load X Minus Immediate
LXP	Load X Positive Immediate
SAI	Subtract from A Immediate
SCM	Scan Memory
SCMB	Scan Memory Byte
SCN	Scan Memory (ALPHA-16)
SXI	Subtract from X Immediate

3.5.2.3 Operand Field

The operand field must contain an expression whose value is in the range:

$$0 \leq \text{Expression} \leq 255$$

Some common types of expressions which appear in Class 2 instructions are:

a. A single ASCII character: 'A'
b. A numeric constant: 155,077, :AB

The following are cautions which must be observed when writing Class 2 expressions:

a. The evaluated expression must not exceed eight binary bit positions. (see Figure 3-5).

b. The expression may not be signed.

c. Indirect and index symbols (* and @) are not allowed.

d. Literals may not be used.

3.5.2.4 Comments field

The comments field is optional.

3.5.3 Class 2 Examples

The following are examples of Class 2 statement and assembled machine language:

STATEMENT	MACHINE LANGUAGE
CAI 150	:C096
CAI :AB	:C0AB
CAI 'A'	:C0C1

3.6 CLASS 3 - CONDITIONAL JUMP INSTRUCTIONS

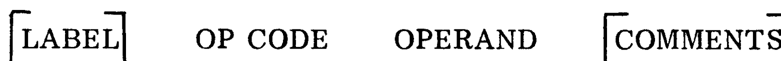
3.6.1 General

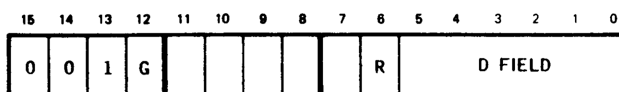
Class 3 instructions are microprogrammed conditional jump instructions. Functions of these instructions are described in subsection 3.5 of the NAKED MINI LSI/ALPHA LSI Programming Reference Manual.

Several of the more commonly used conditional jump conditions are assigned unique op codes which are recognized by the assembler. In addition there is a general op code which allows the programmer to microcode jump conditions which are not defined by unique op codes.

3.6.2 Assembly Format

The general assembly format for Class 3 instructions requires an operation code and an operand expression. The label field and comments field are optional:





Bits	Field	Definition
13-15	Class	Identifies the Conditional Jump Instruction Class
12	G	Test Group Indicator: G=1 for AND Group G=0 for OR Group
7-11	Conditions	Microcode of Test Conditions:
6	R	Jump Direction: R=0 for Forward Jump R=1 for Backward Jump
0-5	D Field	Jump Distance (−63 to +64)

Bit	AND Group	OR Group
7	A Positive	A Negative
8	A≠0	A=0
9	OV Reset	OV Set (Resets OV)
10	SS On	SS Off
11	X≠0	X = 0

Figure 3-6. Class 3 Machine Language Format

3.6.3 Defined Jump Conditions

The following paragraphs describe the assembly format used for writing Class 3 instructions having uniquely defined operation codes.

3.5.3.1 Operation Code Field

The op code field must contain one of the following operation codes:

JAG	Jump if A > 0
JAL	Jump if A ≤ 0
JAM	Jump if A < 0
JAN	Jump if A ≠ 0
JAP	Jump if A ≥ 0
JAZ	Jump if A = 0
JOR	Jump if overflow reset
JOS	Jump if overflow set
JSR	Jump if SENSE switch reset
JSS	Jump if SENSE switch set
JXN	Jump if X ≠ 0
JXZ	Jump if X = 0

3.6.3.2 Operand Field

The operand field is interpreted as the address to which the computer will branch if the jump condition is satisfied. The jump address must be in the range -63 to +64 locations from the instruction.

The following is an example of a backward jump:

```
:200    LOOP    LDA    ABC
:201            ADD    DEF
:202            JAM    LOOP
```

The symbol LOOP has the value :200, which is 2 locations backward from the instruction at :202. The D field of the assembled object code (see figure 3-6) is therefore assembled as:

D=:02

The assembled object code for the instruction is:

:20C2

The following is an example of a forward jump:

```
:250        JAM        $+3
:251         .
:252         .
:253         .
```

The operand is evaluated as:

$ + 3 = :250 + 3 = :253

Forward jumps are relative to P + 1, therefore the value to be placed in the D field is:

:253 - (250 + 1) = :253 - :251 = 2

The assembled object code is therefore:

:2082

3.6.4 Microcoded Jump Conditions

A special operation code, JOC, is provided to allow the programmer to microcode jump conditions. The following paragraphs explain the use of the JOC op code.

3.6.4.1 Format

The format for the JOC op code is:

```
JOC        M,ADDR
```

Refer to figure 3-6. When the JOC op code is recognized by the assembler it causes the operand field to be interpreted as two separate fields.

3.6.4.2 Operand Field

The operand field is written as two separate expressions separated by a comma:

M = An expression representing a bit pattern which is right justified in bits 7-12 of the object code.

ADDR = An expression representing a jump address.

In the condition bits of the M field, a 1-bit means that a test is to be performed, and a 0-bit means that a test is not to be performed.

For example, to code the test: jump if A > 0 (A > 0 means A is a positive and A ≠ 0). Therefore the AND group is selected (G=1), and condition bits 7 and 8 are selected. All other condition bits are set to zero. The required bit pattern in the M field is:

Bits:	12	11	10	9	8	7
	1	0	0	0	1	1

This bit pattern can be represented as :23 or 043. Therefore the statement can be coded as:

```
JOC :23,ADDR
```

The address portion, ADDR, is coded and interpreted in the same manner as for other Class 3 instructions. The assembler sets or resets the R bit (bit 6) and fills in the jump distance according to the location of ADDR relative to the instruction.

3.7 CLASS 4 - SHIFT, SINGLE REGISTER

3.7.1 General

The functions performed by Class 4 instructions are described in subsection 3.6 of the NAKED MINI LSI/ALPHA LSI Programming Reference Manual. Class 4 includes arithmetic shifts and single register logical shifts.

3.7.2 Assembly Format

Figure 3-7 illustrates the assembled object code format for Class 4 instructions. The source code format is as follows:

[LABEL] OP CODE OPERAND [COMMENTS]

The label field and comments field are optional. The Op Code field and Operand field are mandatory.

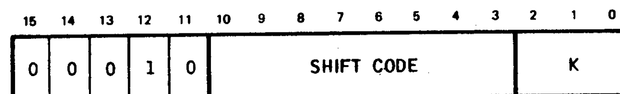


Figure 3-7. Class 4 Machine Language Format

3.7.3 Single Register Shift Instructions

The following paragraphs describe the assembly format used for writing Class 4 shift instructions.

3.7.3.1 Operation Code Field

The op code field must contain one of the following op codes:

ALA	Arithmetic Shift A Left
ALX	Arithmetic Shift X Left
ARA	Arithmetic Shift A Right
ARX	Arithmetic Shift X Right
LLA	Logical Shift A Left
LLX	Logical Shift X Left
LRA	Logical Shift A Right
LRX	Logical Shift X Right
NOR	Normalize (ALPHA-16 Only)
RLA	Rotate A Left
RLX	Rotate X Left
RRA	Rotate A Right
RRX	Rotate X Right

3.7.3.2 Operand Field

The operand field must contain an absolute expression in the range:

$$1 \leq \text{expression} \leq 8$$

Indexing, indirect addressing, and literals are not allowed.

The expression in the operand field represents a shift count. Class 4 shift instructions can perform shifts of from 1 to 8 bit positions. The computer hardware interprets a shift count as 1 + K. Therefore:

$$K = \text{Shift Count} - 1$$

For example:

```
ARA          5
```

The shift count is 5, therefore:

$$K=5 - 1 = 4$$

The assembled instruction is:

:10D4

3.7.4 Bit Manipulation Instructions

The following paragraphs describe the assembly format used for writing class 4 bit manipulation instructions. These instructions are implemented only on the ALPHA LSI computer.

3.7.4.1 Operation Code Field

The op code field must contain one of the following op codes:

BAO	Bit of A to Overflow (ALPHA LSI only)
BXO	Bit of X to Overflow (ALPHA LSI only)

3.7.4.2 Operand Field

The operand field must contain an absolute expression in the range

$$0 \leq \text{expression} \leq 15$$

Indexing, indirect addressing, and literals are not allowed.

The expression in the operand field represents the bit position of the A or X register whose contents are to be moved to the overflow register. Bit 15 corresponds to the left-most bit, bit 0 corresponds to the right-most bit.

Bit manipulation hardware instructions operate from 1 to 8 bit positions from each end of the register. The assembler must therefore reduce the operand to a range from 1 to 8 and generate the applicable instruction. The computer hardware interprets the value as K, therefore:

K = value for $0 \leq$ value ≤ 7
K = 15-value for $8 \leq$ value ≤ 15

For example:

BXO 4

The bit position is 4, therefore:

$K = 4$

The assembled instruction is:

: 13A4

A second example:

BXO 14

The bit position is 14, therefore:

K = 15-14 = 1

The assembled instruction is:

: 1321

3.8 CLASS 7 - LONG SHIFT

3.8.1 General

Class 8 instructions are double-register shift instructions. The functions performed by these instructions are described in subsection 3.6 of the NAKED LSI Programming Reference Manual.

3.8.2 Assembly Format

Figure 3-8 illustrates the assembled machine language format for Class7 instructions. This format differs from Class 4 in that the K field contains 4 bits instead of 3. The source code format is as follows:

[LABEL] OP CODE OPERAND [COMMENTS]

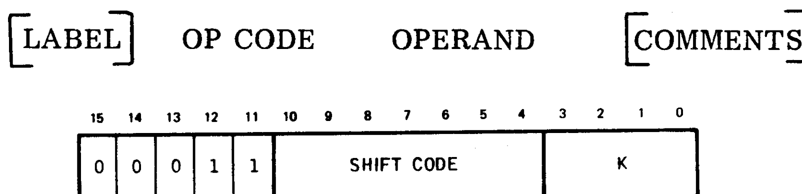


Figure 3-8. Class 7 Machine Language Format

The Label and Comments fields are optional. The Op Code and operand fields are mandatory.

3.8.2.1 Operation Code Field

The op code field must contain one of the following Class 7 op codes:

DVS	Divide Step (ALPHA-16 only)
LLL	Long Logical Left
LLR	Long Logical Right
LRL	Long Rotate Left
LRR	Long Rotate Right
MPS	Multiply Step (ALPHA-16 only)

3.8.2.2 Operand Field

The operand field must contain an absolute expression in the range:

$$1 \leq \text{expression} \leq 16$$

Indexing, indirect addressing, and literals are not allowed.

The expression in the operand field represents a shift count. The computer hardware interprets a shift count as 1 + K, therefore:

K = shift count - 1

For example:

```
LLL     16
```

The machine language K field is interpreted as:

K = shift count - 1 = 16-1=15=:F

The assembled machine language for the above instruction is:

:1B0F

3.9 CLASS 5 - REGISTER CHANGE AND CONTROL

3.9.1 General

The functions of register change instructions and control instruction are explained in subsections 3.7 and 3.8 of the Programming Reference Manual.

3.9.2 Assembly Format

The only mandatory field for source statements for Class 5 instructions is the Operation Code Field. The Label and Comments fields are optional. An operand must not be present:

[LABEL] OP CODE [COMMENTS]

The Op Code field is terminated by a space. Everything between that space and the carriage return which terminates the statement will be interpreted as a comments field.

3.9.2.1 Register Change Op Code

The following are the Class 5 Register Change op codes which are recognized:

ANA	AND of A and X to A
ANX	AND of A and X to X
ARM	Set A to -1
ARP	Set A to +1
AXM	Set A and X to -1
AXP	Set A and X to +1
CAR	Complement A
CAX	Complement A and put in X
CXA	Complement X and put in A
CXR	Complement X
DAR	Decrement A
DAX	Decrement A and put in X
DXA	Decrement X and put in A
DXR	Decrement X
IAR	Increment A
IAX	Increment A and put in X
ISA	Input Data switches to A
ISX	Input Data switches to X
IXA	Increment X and put in A
IXR	Increment X
LAO	Least significant bit of A to OV
LXO	Least significant bit of X to OV
NAR	Negate A
NAX	Negate A and put in X
NRA	NOR of A and X to A
NRX	NOR of A and X to X
NXA	Negate X and put in A
NXR	Negate X
SAO	Sign of A to OV
SXO	Sign of X to OV

TAX	Transfer A to X
TXA	Transfer X to A
XRM	Set X to -1
XRP	Set X to +1
ZAX	Zero A and X
ZAR	Zero A
ZXR	Zero X

3.9.2.2 Control Op Codes

The following are the Class 5 control op codes which are recognized:

CID	Console interrupt disable
CIE	Console interrupt enable
COV	Complement overflow
DIN	Disable interrupts
EAX	Exchange A and X (ALPHA-LSI only)
EIN	Enable interrupts
HLT	Halt
ICA	Input Console to A (ALPHA-LSI only)
ICX	Input Console to X (ALPHA-LSI only)
IPX	Increment P to X (ALPHA-LSI onlv)
NOP	No operation
OCA	Output A to Console (ALPHA LSI only)
OCX	Output X to Console (ALPHA LSI only)
PFD	Power fail interrupt disable
PFE	Power fail interrupt enable
ROV	Reset overflow
SBM	Set byte mode
SIA	Status input to A
SIN	Status inhibit
SIX	Status input to X
SOA	Status output from A
SOV	Set overflow
SOX	Status output from X
SWM	Set word mode
TRP	Trap
WAIT	Wait for Interrupt

3.10 CLASS 6 - INPUT/OUTPUT

3.10.1 General

Input/output instructions are explained in subsection 3.9 of the Programming Reference Manual. Class 6 includes general I/O instructions, block transfer instructions, and automatic I/O instructions.

3.10.2 Assembly Format

The general format for Class 6 instructions is:

[LABEL] OP CODE OPERAND [COMMENTS]

The Op Code and Operand fields must be present. The Label and Comments fields are optional.

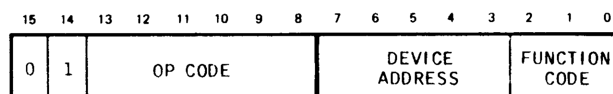


Figure 3-9. Class 6 Machine Language Format

3.10.2.1 Operation Code Field

The Op Code field must contain one of the following Class 6 op codes:

AIB	Automatic input, byte
AIN	Automatic input, word
AOB	Automatic output, byte
AOT	Automatic output, word
BIN	Block input
BOT	Block output
IBA	Input byte to A
IBX	Input byte to X
IBAM	Input byte to A, masked
IBXM	Input byte to X, masked
INA	Input word to A
INX	Input word to X
INAM	Input word to A, masked

INXM	Input word to X, masked
OTA	Output A
OTX	Output X
OTZ	Output zeros
RBA	Read byte to A
RBX	Read byte to X
RBAM	Read byte to A, masked
RBXM	Read byte to X, masked
RDA	Read word to A
RDX	Read word to X
RDAM	Read word to A, masked
RDXM	Read word to X, masked
SEA	Select and present A
SEL	Select function
SEN	Sense and skip on response
SEX	Select and present X
WRA	Write from A
WRX	Write from X
WRZ	Write zero

3.10.2.2 Operand Field

The operand field for Class 6 instructions represents a device address and a function code (see figure 3 - 9). This field may be written in one of two ways:

a. A single absolute expression which represents both the device address and function code:

```
INA        ADDR
```

In this case ADDR must be absolute and in the range:

$$0 \leq ADDR \leq 255$$

b. Two absolute expresions separated by a comma, where the first expression represents a device address and the second a function code:

```
INA        DA, FC
```

In this case DA represents a device address. It must be absolute and in the range:

$$0 \leq DA \leq 31$$

FC represents a function code which must be absolute and in the range:

$$0 \leq FC \leq 7$$

The absolute value of the DA expression is right justified in bits 3-7 of the assembled object code, and the FC expression is right justified in bits 0-2.

Either of these formats may be used in writing an expression. For example, if device 7 and function code 1 are specified with a Select instruction, either of the following statements could be used:

```
a.    SEL  071        (or SEL  :39)
b.    SEL  7,1
```

A symbolic expression may be used, provided it is defined as being absolute:

```
TTY  EQU  :39

     RBA  TTY
```

3.10.3 Multi-word Instructions

Automatic I/O instructions and Block I/O instructions require more than one word in memory. For example, an Automatic Input Byte (AIB) requires three words. The first word defines the instruction and device, the second word specifies the byte count, and the third word identifies the memory buffer area:

```
AIN                DA, FC
BYTE COUNT  (NEGATIVE)
BUFFER ADDRESS -1
```

Only the first word is written and assembled as a Class 6 instruction. The two additional words are generally written as data statements:

```
AIN          7,0
DATA         -80
DATA         BFR-1
```

Section 4

DIRECTIVES

4.1 INTRODUCTION

Directives control the assembly process, establish program relocatability, provide for sub-program linkage, and specify various types of constants, blocks of memory and labels used in the program.

4.2 ASSEMBLER CONTROL DIRECTIVES

The assembler control directives establish and/or alter the value and relocation attribute of the program location counter, provide for conditional assembly of source statements and/or alter the assembly process. Each program should start with an ABS or REL directive and end with an END directive.

In the following descriptions, brackets are used to indicate optional fields. In general a label, if present in the label field, will be assigned the current value and relocation attribute of the program location counter.

4.2.1 Absolute Assembly (ABS)

[LABEL]   ABS   [Expression]

The ABS directive allows the specification of the absolute memory address at which succeeding data/instructions will be loaded and executed. Programs assembled in ABS mode may not be located or executed in any other area of memory.

The ABS directive sets the relocation attribute of the program location counter to absolute (absolute mode of assembly). If an expression is present, the value of the program location counter is set to that value; if an expression is not present, the program location counter value is not altered.

If an expression is present, the expression must have been previously defined; that is, forward references or undefined expressions are invalid and will be flagged by the assembler as being in error. The expression may not be an external symbol.

The label, if present, will be assigned the value of the expression in the operand field.

Absolute assemblies should start with an ABS directive.

4.2.2 Relocatable Assembly (REL)

[LABEL] REL [Expression]

The REL directive allows the specification of the memory address (relative to the base specified at load time) at which succeeding data/instructions will be loaded and executed. Programs assembled in REL mode may be loaded and executed in areas of memory other than those specified at assembly time, by addition of an offset, or base, at load time.

The REL directive sets the relocation attribute of the program location counter to relative (relocatable mode of assembly). If an expression is present, the value of the program location counter is set to that value. If an expression is not present, the program location counter is not altered.

If an expression is present, the expression must have been previously defined; that is, forward references or undefined expression are invalid and will be flagged by the assembler as being in error. The expression may not be an external symbol.

The label, if present, will be assigned the value of the expression in the operand field.

Relocatable assemblies should start with a REL directive.

4.2.3 Origin (ORG)

[LABEL] ORG Expression

The program location counter is set equal to the value of the expression. The mode of the location counter (ABS, REL) is unchanged. The expression must be defined, that is, forward references or undefined expressions are invalid and will be flagged by the assembler as being in error. The expression may not be an external symbol.

The label, if present, will be assigned the value of the expression in the operand field.

Normally, relocatable assemblies should start with ORG 0 or REL 0.

4.2.4 End of Assembly (END)

[LABEL] END [Expression]

The END directive signifies the end of an assembly. Every program or subprogram must terminate with an END directive. The expression, if present, will be interpreted by the object loader as the execution address to transfer to at the end of a successful load. Since the object loader does not distinguish between END directives

in main programs and END directives in subprograms, only the main program END directive should include a transfer address. The object loader will interpret the last transfer address encountered during a load operation as the execution address. If the operand field of the END directive is left blank, no execution address will be output to the object loader.

4.2.5 Conditional Assembly (IFT, IFF, ENDC)

```
[LABEL]     IFT     Expression
[LABEL]     IFF     Expression
[LABEL]     ENDC
```

The IFT (IF True) and IFF (IF False) directives are provided to conditionally assemble subsequent lines of source code. The expression must be absolute (not external or relocatable) and must previously have been defined. The last line affected by IFT/IFF must be an ENDC directive which signals the end of the conditional assembly. IFT/IFF directives may not be nested; that is, another IFT/IFF directive may not appear between an IFT/IFF directive and the associated ENDC directive.

Example:

```
IFT     A
 .
 .
 .
 .
ENDC
```

If the value of A is zero (false), all source statements between the IFT and ENDC directives will be skipped. If the value of A is non-zero (true), the code will be assembled.

Example:

```
IFF     B
 .
 .
 .
ENDC
```

If the value of B is non-zero (true), all source statements between the IFF and ENDC directives will be skipped. If the value of B is zero (false), the code will be assembled.

Example:

```
A       EQU    1
        LDA    TAG1
        IFT    A
        SUB    TAG2
        ENDC
        IFF    A
        ADD    TAG2
        ENDC
        STA    TAG 3
```

The assembler will assemble the above example as if it had been written as follows:

```
        LDA    TAG1
        SUB    TAG2
        STA    TAG3
```

The instruction ADD TAG 2 was not assembled because the IFF A condition was not satisified; that is, A is not false (zero).

If the statement:

```
A       EQU    1
```

were replaced by:

```
A       EQU    0
```

then the above example would be assembled as if it were written as follows:

```
        LDA    TAG1
        ADD    TAG2
        STA    TAG3
```

The END directive, which terminates an assembly, cannot be within the conditional assembly limits (END cannot appear between an IFT/IFF directive and the associated ENDC directive).

4.2.6 Machine Directive (MACH)

[LABEL]   MACH   Expression

The MACH directive allows the user to specify which CAI 16-bit computer's instruction set is to be considered valid during this assembly. This allows the assembly, and/or error detection, of programs written for either (or both) ALPHA-LSI and ALPHA-16

computers. Instructions declared invalid by the MACH directive will be flagged with an "O" error, but will be assembled correctly.

The expression in the operand field must be present, absolute (not relocatable or external), and must be previously defined. The value of the expression will replace the current value in the MACH flag word, remaining in effect until the end of the current assembly or another MACH directive is encountered. The acceptable values of the MACH directive are shown in table 4-1, below.

The label, if present, will be given the current location counter value.

Table 4-1. MACH Flag Word Values

MACH Value	Instruction Set Allowed
0	Subset of ALPHA LSI and ALPHA 16
1	ALPHA 16
2	ALPHA LSI
3	ALPHA LSI and ALPHA 16

4.2.7 Title Directive (TITL)

[LABEL] TITL [ASCII String]

The TTL directive is provided for documentation purposes and will cause the assembler to output a 'top-of-form' to the listing device.

The TITL source record does not appear on the listing, therefore labels should be used with extreme caution since they will not appear on the listing. They will, however, appear in the symbol table.

4.3 DATA AND SYMBOL DEFINITION DIRECTIVES

4.3.1 Reserve Storage (RES)

[LABEL] RES Expression 1 [,Expression 2]

This directive "reserves" a block of storage. The label, if present, will be defined equal to the location of the first reserved word. Expression 1 is the number of words to be reserved. If Expression 1 is symbolic, the symbol must be absolute (not relative or external) and must have been previously defined.

Expression 2, if present is a constant to be stored at each reserved location. If Expression 2 is symbolic, the symbol must be absolute (not relative or external) and must have been previously defined. The object loader will, at load time, store the constant at each of the reserved locations. If Expression 2 is not present, the contents of the reserved memory locations will not be altered at load time.

Example:

```
RES     10
```

Ten words of memory will be reserved and their contents left unaltered. This will have the same effect as if the programmer wrote ORG $+10.

Example:

```
RES     3,:FF
```

Three words of memory will be reserved and the value :FF will be stored at load time, in each of these locations. This will have the same effect as if the programmer wrote three DATA directives.

i.e.

```
RES     3,:FF     is synonomous with
DATA    :FF
DATA    :FF
DATA    :FF
```

4.3.2 Equate (EQU)

```
LABEL       EQU       Expression
```

The EQU directive assigns to a symbol (in the label field) a value other than the one normally assigned by the program location counter. The symbol in the label field must be undefined and will be assigned the value and relocatability of the expression in the operand field.

If the expression is symbolic, it must have previously been defined in the source program. Undefined or external expressions are invalid.

The EQU directive may be used to symbolically equate two locations in memory, or it may be used to give a value to a symbol.

Examples:

A	EQU	15	assigns the value 15 to the symbol A
B	EQU	A	assigns the value of A (15 in this example) to the symbol B.
C	EQU	A+5	assigns the value of A plus 5 (20 in this example) to the symbol C

4.3.3 Set (SET)

```
LABEL    SET    Expression
```

The SET directive is identical to the EQU directive except that the symbol being defined (in the label field) may be re-defined by another SET directive.

Example:

```
A        SET        0
         IFT        A
         LDA        X
         STA        Y
         ENDC
A        SET        1
         IFF        A
         LDA        X
         STA        Y
         ENDC
          .
          .
          .
         etc.
```

The symbol A will first be assigned the value 0 and subsequently be reassigned the value 1. Only symbols defined by a SET directive may be re-defined in this manner without causing a multiply defined symbol error to be noted by the assembler.

4.3.4 Test String (TEXT)

```
[LABEL]    TEXT    'character string'
```

The TEXT directive enables the user to assemble an ASCII character string for use as data. The character string to be assembled must be surrounded by quotes ('). The quote ('), when desired as a character in the string, must appear in the string as two contiguous quotes.

The characters in the expression are assembled two characters per word of memory. Trailing character positions are filled with blanks. The label, if present, will be defined as the location of the first pair of characters assembled.

Examples

```
TEXT     'A' will assemble as C1A0
TEXT     'AB' will assemble as C1C2
TEXT     'ABCD' will assemble as C1C2
                                 C3C4
TEXT     'A''' will assemble as C1A7
```

4.3.5 Data Definition (DATA)

[LABEL] DATA Expression 1 [,Expression 2,......,Expression m]

The operand field of the DATA directive contains one or more expressions, separated by commas. The expressions will be evaluated, one at a time, and generated as constants. The label, if present, will be defined as the location of the first generated constant. Any valid expression may appear in the operand field of a data statement. Allowable expressions include:

```
DATA     10       decimal constant
DATA     077      octal constant
DATA     :FF      hexadecimal constant
DATA     'AB'     character constant (2 characters maximum)
DATA     ABC      symbolic variable
DATA     $        current value of location counter
DATA     A,B,C    multiple constants
```

The rules for the information of expressions apply to the DATA directive. Any valid term may appear and any combination of valid terms jointed by the arithmetic operators + or - may be used.

Example:

```
DATA     $+10
DATA     TABLE+4
```

If the indirect indicator (*) precedes any of the terms in an expression, the most significant bit of that term will be set on. This is useful in generating indirect address pointers.

Example:

```
A        EQU      :100
         DATA     *A
```

The constant generated by the DATA *A directive will be :8100 since the A was preceded by an asterisk.

4.3.6 Byte Address Constant (BAC)

[LABEL] BAC Expression 1 [,Expression 2,...,Expression m]

This directive is used to generate byte address constants.

Example:

```
AA      EQU     :10
A       EQU     :100
B       BAC     A
C       BAC     A+1
D       BAC     A+AA
E       BAC     B,C,D
```

Symbolic items in the variable field are assumed to have "word address" values and numeric items are assumed to be "byte counts" or "byte address" values.

The byte address constant B will be assigned the value :200. Since A has a 'word' value of :100, the word address is doubled by the assembler to create a byte constant of :200.

The byte address constant C will similarly have the value :201 (:100 times 2 plus 1); and the byte address constant D will have the value :220 (:100 + :10 times 2).

4.4 PROGRAM LINKAGE DIRECTIVES

4.4.1 External Name Definition (NAM)

[LABEL] NAME symbol [,symbol,....,symbol]

The NAM directive is used for establishing the necessary linkages between programs that have been assembled separately but are to be loaded and executed together (i.e. external subprograms). The operand field of the NAM directive contains one or more symbols, separated by commas. These symbols are the names or labels defining the entry or reference points into the current program in which reference is made by external programs.

Each name appearing in the operand field of the NAM directive must also appear in the label field of a statement in the body of the program. The NAM directive, when used, must precede all other statements except comment statements, user-defined op-code definition statements, or TITL directives.

For example, an external subroutine to evaluate SIN and COS might be written to be referenced but not assembled by the calling program. The NAM directive would be required in this case to define the two entry points to the subprogram as follows:

```
        NAM   SIN
        NAM   COS
SIN     ENT
         .
        RTN   SIN
         .
         .
COS     ENT
         .
        RTN   COS
        END
```

4.4.2 External Reference (EXTR)

[LABEL]   EXTR   symbol [,symbol....,symbol]

The EXTR directive is used to declare references to all external symbols (not defined in the current program) referenced by the program being assembled. Linkage, at load time, to these declared external symbols will be made through scratchpad, by the object loader.

Each name or symbol appearing in the operand field of the EXTR directive will be output to the object loader providing that name is referenced within the program being assembled. Since these names are not defined within the current program, they may not be used in multi-termed expressions.

For example, a program that will call an external subprogram named SIN might contain the following EXTR directives:

```
REL    0
EXTR SIN
  .
  .
JST    SIN
  .
  .
END
```

4.4.3 External Reference (REF)

```
        LABEL    REF
```

This directive is very similar to EXTR; that is, it is used to declare references to symbols that are referenced by the current program but are external to the current program.

The primary difference between REF and EXTR is that the REF directive defines the location in memory to be used for linkage to the external subprogram while EXTR causes the linkage to be made through scratchpad.

For example, a program that calls an external subprogram SIN might contain the following REF directive:

```
        REL   O
         .
         .
        JST   *SIN
         .
         .

SIN     REF
```

At load time, the address at which the SIN subprogram was loaded would be stored in the memory location indicated by the REF directive.

4.5 PSEUDO INSTRUCTION

4.5.1 Subroutine Call (CALL)

```
        [LABEL]    CALL    Symbol
```

This directive causes the assembler to generate a Jump-and-Store instruction to the symbol in the operand field (JST Symbol). It is provided primarily for documentation purposes to facilitate recognition of subroutine call instructions.

Example:

```
        CALL    SUB
```

4.5.2 Subroutine Entry (ENT)

```
LABEL         ENT
```

This directive causes the assembler to reserve a word to be used to hold the return address from a subroutine call. The assembler generates a HLT instruction for this directive.

Example:

```
SIN     ENT
         .
         .
         .
         .
        RTN     SIN
```

The ENT directive is provided for documentation purposes primarily. Any source statement that will result in one word being reserved may be used in its place. For example, the SIN ENT statement could have been any of the following statements instead:

```
SIN     HLT
SIN     NOP
SIN     DATA    0
SIN     RES     1
```

4.5.3 Subroutine Return (RTN)

```
[LABEL]       RTN       symbol
```

This directive causes the assembler to generate an indirect (Jump) instruction via the symbol in the operand field (JMP *symbol). It is provided primarily for documentation purposes to facilitate recognition of subroutine return instructions.

Example:

```
SIN     ENT
         .
         .
         .
         .
        RTN     SIN
```

4.5.4 Coded Halt (STOP)

[LABEL] STOP Expression [COMMENTS]

This directive causes the assembler to generate a halt (HLT) instruction with the expression field imbedded in the low-order 8 bits. It is provided to allow the display of various coded values in the computer Instruction (I) register when the program must terminate execution.

The expression in the operand field must be present, absolute (not relocatable or external), and must be previously defined. The value must be in the range :00 to :FF and will be inserted into the low-order byte of the instruction.

The label, if present, will be given the current value of the location counter.

Example:

```
STOP   :10
```

4.5.5 Wait for Interrupt (WAIT)

[LABEL] WAIT [COMMENTS]

This directive causes the generation of a Jump-to-Self instruction (JMP $). Provided primarily for documentation purposes, it allows the program to be put in an indefinite loop to await the occurrence of an interrupt.