Section 5

USER-DEFINED OPERATION CODES

5.1 INTRODUCTION

This feature allows the user to name or define his own instruction mnemonics for the current assembly. User-defined operation code definitions, if used, must precede all source statements except comment statements and TITL directives.

5.2 FORMAT

One source statement is required for each operation code definition. The format of "op-def" lines is as follows:

5.2.1 Label Field

The first character of the source statement must be a dollar sign ($). The instruction type code immediately follows the dollar sign (see type code table below). A blank character(s) may not appear between the dollar sign and the type code. One or more blank (space) characters delimit the label field. The type code will tell the assembler how to assemble the op-code being defined; that is, which assembly class does this new op-code fall into for assembly purposes.

5.2.2 Operation Code Field

The mnemonic operation code being defined must appear in this field. Valid operation codes consist of from 1 to 4 characters. Any character with the exception of a space (blank) may be used to form the mnemonic operation code.

5.2.3 Expression or Operand Field

The operand field contains the "instruction skeleton" to be used when assembling statements containing user-defined operation codes. The expression must be a hexadecimal number.

5.2.4 Comments Field

The comments field is ignored by the assembler (except for listing purposes) and is available for comments as with any other source statements.

5.3 RESTRICTIONS

The operation codes being defined will be retained by the assembler for the duration of the current assembly only.

If an op-code is defined that is in the assembler's standard op-code table, the new operation will be the one used for the current assembly.

Directives cannot be re-defined.

5.4 INSTRUCTION TYPE CODE TABLE

User-defined operation codes will be assembled in the same manner as the standard operation codes. The user must select one of the classes of instructions into which the defined operation code will fit. Selecting one of the classes effectively selects the assembly syntax for the defined op-code. For example, Class 5 instructions do not require an operand. The "instruction skeleton" is output as the asembled instruction.

CLASS/CODE	INSTRUCTION TYPE
1	Memory reference instructions, Word.
2	Immediate-address instructions.
3	Conditional jump instructions.
4	Shift instructions (single register).
5	Micro instructions.
6	Input-Output instructions.
7	Shift instructions (double register).
8	Memory reference instructions, Byte.
9	Memory reference instructions, Double Word

5.5 USAGE

5.5.1 Example 1

Many of the available conditional jump instructions have not been given unique mnemonics. The user may name these instructions using the operation code definition capability as follows:

```
$3      ANZ     :2180
```

This will assign the mnemonic ANZ to the conditioned jump instruction "Jump if A register is negative or zero". The instruction may then be written:

```
ANZ     ABC
```

which will cause a jump to location ABC if the A register is negative or zero.

The same instruction could be written, using the generalized JOC op-code as follows:

```
JOC     :3,ABC
```

5.5.2 Example 2

The following instruction will cause the character in the A register to be typed on the ASR-33 printer:

```
WRA     :3B
```

The user could write the following operation code definition to define an instruction called TYPE:

```
$5      TYPE    :6D3B
```

This will define "TYPE" as a Class 5 instruction (one which does not require an operand) and the programmer could then cause the character in the A register to be printed by writing TYPE in the op-code field of an instruction.

```
i.e.    LAP 'A'
        TYPE    Type the character A
```

Section 6

ASSEMBLER OUTPUTS

6.1 INTRODUCTION

Figure 1-1 illustrates the source input and the two outputs of the Beta Assembler. This section explains the assembler outputs.

6.2 SOURCE INPUT

6.2.1 Source Program

Figure 6-1 illustrates a typical source program as it may appear on a programmer's coding sheet. The source code is keypunched into a free form paper tape for input to the Beta Assembler.

6.2.2 Source Tape

The punched paper tape that is generated from the source code is called a source tape. The tape is unformatted, with one or more spaces separating fields and a carriage return (CR) delimiting each line.

6.3 ASSEMBLY LISTING

6.3.1 General

Figure 6-2 illustrates an assembly listing which was generated when the program shown in Figure 6-1 was assembled. Although the source input tape was unformatted, the assembly listing is formatted.

6.3.2 Listing Format

The following paragraphs explains the assembly listing. The listing in Figure 6-2 is used as an illustration.

```
*
*
*       SAMPLE ASSEMBLY LISTING.
*
*       THE FOLLOWING IS A SAMPLE ASSEMBLY
*       LISTING FROM THE BETA ASSEMBLER.
*
*
```

```
          EXTR      SUBR      EXTERNAL SUBROUTINE REF.
          REL       :100      SET LOC CTR AND RELATIVE.
START     LDA       B         SINGLE LINE OF OUTPUT.
          LDA       =100      TWO LINES OF OUTPUT.
A         EQU       :F5       NO LOCATION LISTING.
          TEXT      'ABC'     TEXT STRING, TWO LIST LINES.
B         DATA      0         DATA STATEMENT, ONE LINE.
C         DATA      1,2,3     DATA, MULTIPLE LINES.
          LDA       E         REF TO UNDEFINED SYMBOL.
          JST       SUBR      REF TO EXTERNAL SYMBOL.
          ORG       D         INVALID FORWARD REF.
D         LDA       AB        REF TO MULT DEFINED SYMBOL.
AB        DATA      500       MULTIPLY DEFINED SYMBOL.
AB        DATA      A,B,C     MULTIPLY DEFINED SYMBOL.
          END       START     END WITH TRANSFER ADDR.
```

Figure 6-1. Source Code

6.3.2.1 Page Numbers

Each page of the assembly listing is headed by a page number. In this example, Page 1 contains a listing of the assembled source code, and Page 2 contains a listing of the symbol table.

6.3.2.2 Line Number

Column 1 of the listing contains the source statement line number in decimal. Each source statement is assigned a line number by the assembler. Note that a single statement may generate more than one line of assembler listing text (see lines 12, 13, 16, and 22).

```
PAGE 0001

0001                    *
0002                    *
0003                    *       SAMPLE ASSEMBLY LISTING.
0004                    *
0005                    *       THE FOLLOWING IS A SAMPLE ASSEMBLY
0006                    *       LISTING FROM THE BETA ASSEMBLER.
0007                    *
0008                    *
0009                              EXTR    SUBR    EXTERNAL SUBROUTINE REF.
0010    0100                      REL     :100    SET LOC CTR AND RELATIVE.
0011    0100    B203    START     LDA     B       SINGLE LINE OF OUTPUT.
0012    0101    B000              LDA     =100    TWO LINES OF OUTPUT.
                0064
0013            00F5    A         EQU     :F5     NO LOCATION LISTING.
0014    0102    C1C2              TEXT    'ABC'   TEXT STRING.  TWO LIST LINES.
        0103    C3A0
0015    0104    0000    B         DATA    0       DATA STATEMENT.  ONE LINE.
0016    0105    0001    C         DATA    1,2,3   DATA, MULTIPLE LINES.
        0106    0002
        0107    0003
0017    0108    B000              LDA     E       REF TO UNDEFINED SYMBOL.
**      U
0018    0109    F900              JST     SUBR    REF TO EXTERNAL SYMBOL.
0019    0000                      ORG     D       INVALID FORWARD REF.
**      U
0020    0000    B000    D         LDA     AB      REF TO MULT DEFINED SYMBOL.
**      D
0021    0001    01F4    AB        DATA    500     MULTIPLY DEFINED SYMBOL.
**      M
0022    0002    00F5    AB        DATA    A,B,C   MULTIPLY DEFINED SYMBOL.
**      M
        0003    0104
        0004    0105
0023            0100              END     START   END WITH TRANSFER ADDR.
0005    ERRORS
```

Figure 6-2. Assembly Listing

6.3.2.3 Location Counter

Column 2 contains the location counter value in hexadecimal, which is associated with each appropriate source statement.

There are no location counter values associated with comment statements, therefore there are no entries in Column 2 for source lines 1-8. Any statement which does not alter the location counter does not have any entry in Column 2 (see lines 9, 13 and 23).

Some statements generate a word of storage which is not to be stored sequentially. For example, line 12 generates a literal value. The literal value will be stored in scratchpad. There is no location counter value associated with the literal value.

Statements which generate more than one sequential word of storage will have a location counter value for each word required (see lines 14, 16, and 22).

6.3.2.4 Assembled Data

Column 3 contains the assembled data represented as four hexadecimal characters for each word. In general, the entries in this column represent data that will be stored in memory at load time.

Refer to line 11. The code B203 is the assembled machine language for the LDA B instruction.

Refer to line 12. Two data words were generated in Column 3 by the one source statement. The code B000 is the assembled instruction. And 0064 is stored in a word in scratchpad by the loader. The loader will place the address of that word in the D field of the assembled instructions.

Refer to line 13. In this case, the entry in Column 3 is the value given to the label by the EQU directive. This value will not be stored in memory at load time.

TEXT and DATA directives may generate more than one line of data in Column 3. Refer to lines 14, 16, and 22 for examples.

Line 23 is the END statement. The number in Column 3 of line 23 represents a transfer address. It is the value of the expression in the operand field of the END directive.

6.3.2.5 Source Code Listing

The remainder of the listing on Page 1 is the formatted source program.

6.3.3 Source Program Errors

During assembly, the assembler checks the source program for syntactic and semantic errors. Whenever an error condition is detected, an appropriate flag is set and the assembly process continues. When the assembly listing is printed, error messages are contained in the listing.

6.3.3.1 Error Messages

When a source error is detected, the assembler prints an error message consisting of two asterisks (**) in Column 1 of the listing and one or more error symbols in Column 2. Lines 17, 19, 20, 21, and 22 contain errors. An error message follows each line.

6.3.3.2 Error Symbols

The following are the error symbols and their meanings:

Symbol	Meaning
A	Address out of range (conditional jumps or immediate instructions)
C	Conditional assembly error (missing ENDC or skipped END)
D	Duplicate symbol reference
E	Expression error (value is assumed to be 0)
L	Syntax error in label field or missing label
O	Operation mnemonic (assumed DATA)
OV	Symbol table overflow
P	Pass 2 out of phase from Pass 1
M	Multiple symbol definition
R	Relocation error
S	Syntax error in operand field
T	Truncation
U	Undefined symbol referenced

To facilitate coding of binary relocatable programs, out-of-range memory reference instructions may optionally be flagged as 'A' errors by the assembler. This option is selected by setting the SENSE switch.

6.3.3.3 Error Count

There is a count of the total number of errors encountered during the assembly.

6.3.3.4 Error Examples

The errors shown in Figure 6-2 are explained below:

Line 17. The error symbol is U, for an undefined symbol. In the instruction

```
LDA     E
```

the symbol E is not defined in the label field.

Line 19. This is also an undefined symbol error, but in this case the symbol D is defined on the next line. This is an error because symbols used with the ORG directive (and with REL and ABS) must be defined in the label field before they are referenced in the operand field.

Line 20. The error symbol is D. In this case the instruction

```
LDA     AB
```

references a multiply defined symbol.

Lines 21 & 22 The symbol AB is used twice in the label field, therefore it is multiply defined. The error symbol is M.

6.4 OBJECT OUTPUT

In the standard system, the Object Output is generated as a punched paper tape, which is referred to as an Object Tape.

A complete description of the format of an object tape is contained in the Object Loader description. This document is the ALPHA LSI LAMBDA Object Loader, Program number 96003.

Section 7

OPERATING PROCEDURES

7.1 INTRODUCTION

7.1.1 General

The Beta Assembler is a computer program which must be loaded into an ALPHA series computer for execution. This section describes the step-by-step procedures which must be followed in assembling a program.

7.1.2 Assumptions

The operating procedures described below make the following assumptions:

a. The source program has been keypunched into a source tape.

b. The operator is familiar with the operation of the computer.

c. The operator is familiar with the operation of the Object Loader (LAMBDA) and the Binary Loader (BLD).

7.2 STEP-BY-STEP PROCEDURES

The following are the step-by-step operating procedures to be followed when assembling a program:

a. Load the Beta Assembler into memory using the LAMBDA Object Loader (for an object tape), BLD or AUTOLOAD.

b. Enter the Beta Assembler at location : 100.

c. The program will halt (I = : 0800) to allow selection of assembly options. Options are selected by entering the appropriate value into the Console Sense Register (low-order Data Switches on ALPHA-16). The options available are:

Listing Device / Punch Device	TTY		Line Printer	
	Complete Listing	Error Only	Complete Listing	Error Only
TTY	: 0	: 1	: 2	: 3
HSP	: 4	: 5	: 6	: 7

To repeat pass 2, add : 8 to the appropriate value (set Data Switch 3 on ALPHA-16).

To flag out-of-range memory reference instructions as "A" errors on listing, set SENSE switch. Normal indirect linkage will be generated.

Reader Control Relay To "IN"

d. Ready the source tape in the TTY paper tape reader or the high speed paper tape reader with the tape leader in the read station: Put the paper tape reader on line. (The assembler will read from whichever paper tape reader device is ready.)

e. Depress RUN to begin the assembly.

f. During the assembly process the Beta Assembler may print messages requiring operator action. The messages and required operator action are as follows:

MESSAGE	ACTION REQUIRED
"FEED ME"	a. Remount source tape in reader and put the read on-line. b. Depress RUN (The source program was too large to be saved in memory and must be reread for subsequent passes.
"PUNCH ON. AT HALT OFF"	a. Turn on TTY punch and depress RUN. b. The assembler will punch the Object Tape and will halt. At the halt, turn the TTY punch off, and depress RUN. (This message will appear only if TTY is used both for listing and punching the object tape.)

MESSAGE	ACTION REQUIRED
"PAUSE"	This message indicates that an up arrow (↑) was encountered in the source input. a. Mount the next source tape segment in the paper tape reader and ready the reader. b. Depress RUN.

g. At the end of the current assembly, the assembler will type "BETA" and will halt. Go back to step 3 to perform another assembly.

7.3 ALTERATION OF ASSEMBLER VARIABLES

The ability to operate the Assembler under non-standard conditions is available through the modification of certain memory locations in lower base page. If it is desired to operate in this condition on a permanent basis, the modifications can be made and the assembler dumped on paper tape with the Binary Dump (BDP) program.

The options available, their absolute memory address and default values are shown in table 7-1 below.

Table 7-1. Assembly Variables

Absolute Location	Value	Function
:0002	Variable	Highest memory location available to BETA Assembler; computed upon initial entry to BETA. To alter this address (to preserve utilities, etc.) alter location :0002 and NOP location :0100.
:0003	:0002	Machine (MACH Directive) value; default is LSI instruction set.
:0004	:FFCB(-53)	Negative of maximum lines/page on listing device. Default allows 13 lines for top and bottom page formatting.
:0005	:FFB0(-80)	Negative of maximum characters/line on listing device.

Appendix A

INSTRUCTION MNEMONICS

Instruction Mnemonic	Description
ADD	Add to A
ADDB	Add byte to A
AAI	Add to A immediate (ALPHA LSI only)
AIB	Automatic Input Byte
AIN	Automatic Input
ALA	Arithmetic shift A left
ALX	Arithmetic shift X left
ANA	AND of A and X to A
AND	AND to A
ANDB	AND byte to A
ANX	AND of A and X to X
AOB	Automatic Output Byte
AOT	Automatic Output
ARA	Arithmetic shift A right
ARM	Set A register to minus 1
ARP	Set A register to plus 1
ARX	Arithmetic shift X right
AXI	Add to X immediate
AXM	Set A and X register to minus 1
AXP	Set A and X register to plus 1
BAO	Bit of A to OV (ALPHA LSI only)
BIN	Block Input to memory
BOT	Block output from memory
BXO	Bit of X to OV (ALPHA LSI only)
CAI	Compare to A immediate
CAR	Complement A register
CAX	Complement A and put in X
CID	Console interrupt disable
CIE	Console interrupt enable
CMS	Compare and skip if high or equal
CMSB	Compare byte and skip if high or equal
COV	Complement overflow
CXA	Complement X and put in A
CXI	Compare to X immediate
CXR	Complement X register
DAR	Decrement A register

Instruction Mnemonic	Description
DAX	Decrement A and put in X
DIN	Disable interrupts
DVD	Divide (ALPHA LSI only)
DVS	Divide step (ALPHA-16 only)
DXA	Decrement X and put in A
DXR	Decrement X register
EAX	Exchange A and X (ALPHA LSI only)
EIN	Enable interrupts
EMA	Exchange memory and A
EMAB	Exchange bytes; memory and A
HLT	Halt
IAR	Increment A register
IAX	Increment A and put in X
IBA	Input byte to A register (unconditionally)
IBAM	Input byte to A register masked (unconditionally)
IBX	Input byte to X register (unconditionally)
IBXM	Inpute byte to X register masked (unconditionally)
ICA	Input Console Data Register to A (ALPHA LSI only)
ICX	Input Console Data Register to X (ALPHA LSI only)
IMS	Increment memory and skip on zero results
INA	Input to A register
INAM	Masked input to A register
INX	Input to X register
INXM	Masked input to X register
IOR	Inclusive OR to A
IORB	Inclusive OR byte to A
IPX	Increment P and put in X (ALPHA LSI only)
ISA	Input data switches to A
ISX	Input data switches to X
IXA	Increment X and put in A
IXR	Increment X register
JAG	Jump if A greater than zero
JAL	Jump if A less than or equal to zero
JAM	Jump if A negative
JAN	Jump if A not zero
JAP	Jump if A positive
JAZ	Jump if A zero
JMP	Jump unconditionally
JOC	Jump on condition
JOR	Jump if OV reset
JSR	Jump if SS off
JSS	Jump if SS on
JST	Jump and store
JXN	Jump if X non-zero
JXZ	Jump if X zero
LAM	Load A minus immediate

Instruction Mnemonic	Description
LAO	Least significant bit of A to OV
LAP	Load A positive immediate
LDA	Load A
LDAB	Load byte to A
LDX	Load X
LDXB	Load byte to X
LLA	Logical shift A left
LLL	Long logical left
LLR	Long logical right
LLX	Logical shift X left
LRA	Logical shift A right
LRL	Long rotate left A and X
LRR	Long rotate right A and X
LRX	Logical shift X right
LXM	Load X minus immediate
LXO	Least significant bit of X to OV
LXP	Load X positive immediate
MPS	Multiply step (ALPHA 16 only)
MPY	Mutiply (ALPHA LSI only)
NAR	Negate A register
NAX	Negate A and put in X
NOP	No operation
NOR	Normalize (ALPHA16 only)
NRM	Normalize (ALPHA LSI only)
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 register
OCA	Output A to Console Data Register (ALPHA LSI only)
OCX	Output X to Console Data Register (ALPHA LSI only)
OTA	Output A register
OTX	Output X register
OTZ	Output zero
PFD	Power Fail interrupt disable
PFE	Power Fail interrupt enable
RBA	Read byte to A register
RBAM	Read byte to A register masked
RBX	Read byte to X register
RBXM	Read byte to X register masked
RDA	Read word to A register

Instruction Mnemonic	Description
RDAM	Read word to A register masked
RDX	Read word to X register
RDXM	Read word to X register masked
RLA	Rotate A left with OV
RLX	Rotate X left with OV
ROV	Reset overflow
RRA	Rotate A right with OV
RRX	Rotate X right with OV
SAI	Subtract A immediate (ALPHA LSI only)
SAO	Sign of A to OV
SBM	Set byte mode
SCN	Scan memory, indexed, indirect (ALPHA-16 only)
SCM	Scan memory, indexed, indirect (ALPHA LSI only)
SCMB	Scan memory byte, indexed, indirect (ALPHA LSI only)
SEA	Select and present A
SEL	Select function
SEN	Sense and skip on no response
SEX	Select and present X
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
SSN	Sense and skip on no response
STA	Store A
STAB	Store byte from A
STX	Store X
STXB	Store byte from X
SUB	Subtract from A
SUBB	Subtract byte from A
SWM	Set word mode
SXI	Subtract from X immediate (ALPHA LSI only)
SXO	Sign of X to OV
TAX	Transfer A to X
TRP	Trap
TXA	Transfer X to A
WRA	Write from A register
WRX	Write from X register
WRZ	Write zeros
XOR	Exclusive OR to A
XORB	Exclusive OR byte to A

Instruction Mnemonic	Description
XRM	Set X register to minus 1
XRP	Set X register to plus 1
ZAR	Zero A register
ZAX	Zero A and X register
ZXR	Zero X register

Appendix B

DIRECTIVE MNEMONICS

Directive Mnemonic	Description
ABS	Absolute Assembly
BAC	Byte Address Constant
CALL	Subroutine Call
DATA	Data Definition
END	End of Assembly
ENDC	End Conditional Assembly
ENT	Subroutine Entry
EQU	Equate
EXTR	External Reference
IFF	Conditional Assembly, If False
IFT	Conditional Assembly, If True
MACH	Set Machine Assembly Mode
NAM	External Name Definition
ORG	Origin
REF	External Reference
REL	Relocatable Assembly
RES	Reserve Storage
RTN	Subroutine Return
SET	Set
STOP	Stop
TEXT	Text String
TITL	Title
WAIT	Wait for interrupt

BETA ASSEMBLER REFERENCE MANUAL
(Supplement)

LSI-2 EXTENDED INSTRUCTIONS

1.0 INTRODUCTION

The NAKED MINI/ALPHA LSI-2 supports an extended set of instructions not found in the LSI-1 or ALPHA 16 computers. This document describes their operation in the syntax of the BETA assemblers (BETA 4/8 and OMEGA), and assumes the user is familiar with the BETA 4 Assembler Reference Manual (document 96018-00).

These instructions are supported in all BETA Assemblers beginning with version -D0 and are made available through use of the MACH directive (described in the BETA 4 Assembler Reference Manual) as follows:

MACH Value*	Instruction Set Allowed
0	Common subset of ALPHA 16 and LSI only
1	ALPHA 16
2	LSI
3	ALPHA 16 and LSI
4	Extended LSI-2
5	ALPHA 16 and Extended LSI-2
6	LSI and Extended LSI-2
7	ALPHA 16, LSI and Extended LSI-2

* Default value of 2 is assumed if no MACH directive is entered.

MACH directives should appear prior to program instructions.

The common subset of ALPHA 16 and LSI instructions is always allowed.

2.0 LSI-2 EXTENDED INSTRUCTIONS

2.1 Instruction Syntax

2.1.1 General

For assembly purposes the LSI-2 extended instructions are divided into two classes.

2.1.2 Instruction Classes

The LSI-2 extended instructions are discussed in this section in a logical sequence rather than numerical sequence. The instruction classes and their sequence of discussion are as follows:

Class 10: Stack, Double Word
Class 5: Register Change and Control

2.1.3 Syntax Description

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

2.2 Class 10 - Stack, Double Word

2.2.1 General

The combination of post-autoincrement addressing in which the stack pointer is stepped toward higher memory after the operand address is determined, and pre-autodecrement addressing in which the stack pointer is stepped toward lower memory before the operand address is determined, is the basic requirement for convenient low overhead stack operations.

The LSI-2 has extensive stack processing capabilities allowing, for example, the nested handling of interrupts and/or subroutine calls. Elements in the stacks may be accessed through indexed addressing. This provides for convenient access of dynamically assigned temporary storage, especially useful in nested procedures. The stack pointer may be manipulated without accessing the stack to allow convenient boundary condition testing.

Stack instructions require two consecutive words of memory, a word for stack pointer, and one or more consecutive words for the stack itself. Addressing modes include direct, indexed, pre-autodecrement (push), and post-autoincrement (pop).

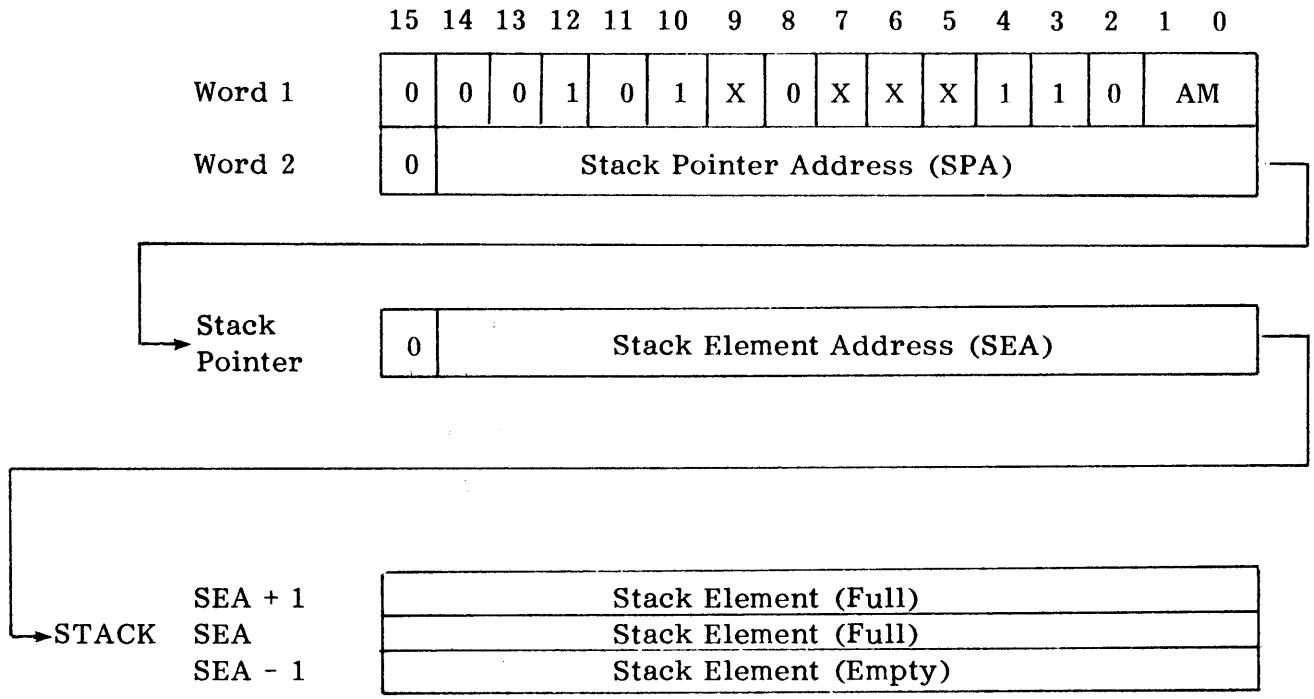

Figure 1 - Class 10 Machine Language Format

Where:
X - Op Code
AM - Addressing Mode
SPA - Stack Pointer Word Address; indirection not allowed.
SEA - Stack Element Word Address; indirection not allowed.

Applicable Addressing Modes (AM) are:

00 - Direct Access to Stack
01 - Indexed Access to Stack
10 - Post-Autoincrement Access to Stack (POP)
11 - Pre-Autodecrement Access to Stack (PUSH)

2.2.2 Assembly Format

Class 10 instruction format for assembly purposes is as follows:

[LABEL] OP CODE OPERAND[,AM] [COMMENT]

2.2.2.1 Label Field. The Label field is optional with Class 10 instructions.

2.2.2.2 Operation Code Field. The Operation Code (Op Code) must be present. Legal Op Codes for Class 10 instructions are:

ADDS	Add Stack Element to A
ANDS	Logical AND Stack element with A
CMSS	Compare Stack element and A
EMAS	Exchange Stack element and A
IMSS	Increment Stack element and Skip on zero
IORS	Inclusive OR Stack element with A
JMPS	Jump to Stack element
JSTS	Jump and Store in Stack element
LDAS	Load A from Stack element
LDXS	Load X from Stack element
SLAS*	Stack Pointer to A
STAS	Store A to Stack element
STXS	Store X to Stack element
SUBS	Subtract Stack element from A
XORS	Exclusive OR Stack element with A

*Note: SLAS does not access a Stack element, but the STACK POINTER (SPA).

2.2.2.3 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.

The second expression (AM) is optional and, when included, must be separated from the first by a comma. This expression represents the addressing mode of the Stack instruction. The following is a list of valid expression characters and their associated addressing modes.

Character	Address Mode
No second expression	DIRECT. Stack element is accessed through Stack Pointer. The Stack Pointer is unchanged.
-	PUSH. Stack Pointer is DECREMENTED; Stack element is then accessed through Stack Pointer.
+	POP. Stack element is accessed through Stack Pointer; Stack Pointer is then INCREMENTED.
@	INDEXED. The sum of the Stack Pointer and index register form the effective address of the Stack element to be accessed.

2.2.2.4 Comments Field. The comments field is optional.

2.2.2.5 Class 10 Examples. The following are examples of Class 10 instructions:

Example 1 - This example illustrates a save/restore sequence using the Stack capability, allowing convenient coding of re-entrant or recursive routines. This example assumes interrupts were disabled by the JST instruction which caused control to be passed to this routine.

```
SUBR    ENT
        STAS    PTR,-       Push 'A' on Stack
        STXS    PTR,-       Push 'X' on Stack
        SIA                 Get CPU status
        STAS    PTR,-       Push on Stack
        LDA     SUBR        Get Return Address
        STAS    PTR,-       Push on Stack
        EIN                 Restore Interrupts
         .
         .
        SIN     6           Disable interrupts
                            during restore
        LDAS    PTR, +      Pop return
        STA     SUBR        and save.
        LDAS    PTR, +      Pop status
        SOA                 and restore.
        LDXS    PTR, +      Pop 'X'
        LDAS    PTR, +      Pop 'A'
        JMP     *RTN        Return
```

Example 2 - This example illustrates an indexed Stack move of 100 entries from Stack 1 to Stack 2, while simultaneously zeroing Stack 1.

```
        LXP     100         Count to move
LOOP    ZAR                 Zero out buffer 1
        EMAS    PTR1,@      Get data (indexed)
        STAS    PTR2,@      Put data (indexed)
        DXR                 Decrement count and Pointer
        JXN     LOOP        Loop back 99 more
         .
         .
PTR1    DATA    STACK1 - 1  Pointer to Stack 1
PTR2    DATA    STACK2 - 1  Pointer to Stack 2
STACK 1 RES     100         Stack 1
STACK 2 RES     100         Stack 2
```

2.3 Class 5 - Register Change and Control

2.3.1 General

The functions of register change instructions and control instructions are explained in subsections 3.7 and 3.8 of the BETA 4 Assembler Reference Manual (document 96018-00).

2.3.2 Assembly Format

The only mandatory field for Class 5 instructions is the Operation Code Field. The Label and comment fields are optional.

[LABEL] OP CODE [COMMENTS]

2.3.2.1 Register Change and Control Op Codes. The following are the extended Class 5 register change and control instructions which are supported on LSI-2.

BCA	Bit Clear A. The contents of the X register are logically complemented and then ANDed with A. The original value of X is left unchanged and the result is left in A.
BCX	Bit Clear X. The contents of the X register are logically complemented and then ANDed with A. The original value of A is left unchanged and the result is left in X.
BSA	Bit Set A. The contents of the X register are logically ORed with A. The original value of X is left unchanged and the result is left in A.
BSX	Bit Set X. The contents of the A register are logically ORed with X. A is left unchanged and the result is left in X.

EIX — Execute Instruction pointed to by X. The instruction whose address is contained in the X register is executed as though it occupied the location following the EIX instruction. The location following the EIX instruction is skipped during execution of the EIX instruction.

If the executed instruction:

1. Is multi-word instruction, the second and succeeding words of the instruction must be located at the second location after the EIX instruction (EIX+2).

2. Modifies the program counter, the modification is relative to location EIX+2.

3. Is a SCM or conditional I/O instruction, the location following the EIX instruction (EIX+1) should be coded with a JMP $-1. This is required for recovery purposes in the event of an interrupt or the lack of a true sense response.

Note that EIX is not interruptable.

2.3.2.2 Class 5 Examples. The following are examples of Class 5 instructions:

Example 1 - This example shows how a single mask word can be used to set or clear one or more flag bits in a flag word.

Setting Bits

```
      .
      .
LDX   MASK    Mask bits to X
LDA   FLAG    Flag word to A
BSA           Set bits in flag word
STA   FLAG    Store new flag word
      .
      .
```

Clearing Bits

```
          .
          .
          LDX       MASK      Mask bits to X
          LDA       FLAG      Flag word to A
          BCA                 Clear bits in flag word
          STA       FLAG      Store new flag word
          .
          .
```

MASK word - Contains "1"s in those bit positions which are to be set or cleared.

Example 2 - This example illustrates how the EIX instruction could be used in a universal output driver, where the I/O commands of each particular device are contained in tabular form, i.e., in tables ordered by logical unit number.

```
                                    X contains the character to be output
                                    A contains the logical unit no.
          ADD       IOINST    Add table address
          EAX                 Address to X, character to A
          EIX                 Execute OTA instruction
          JMP       $-1       Required for conditional I/O
          .
          .
          .
IOINST    DATA      $+1       I/O Table, ordered by logical unit
          OTA       DAX0, FCX0
          OTA       DAX1, FCX1
          OTA       DAX2, FCX2
          .           .     .
          .           .     .
          .           .     .
          OTA       DAXn, FCXn
```