CHAPTER 4

USING THE ASSEMBLER/DISASSEMBLER

Integrated into the MC68000 Educational Computer firmware is an assembler/disassembler function. The disassemble function is called as an option to the Memory Display (MD ;DI) and Memory Modify (MM ;DI) commands, and also is used during execution of system trace and the display register command. The assemble function allows code entry and editing and is invoked by the Memory Modify (MM ;DI) command. Chapter 4 is a detailed discussion of the assembler/disassembler.

CHAPTER 4

USING THE ASSEMBLER/DISASSEMBLER

4.1 INTRODUCTION

Included as part of the MC68000 Educational Computer firmware is an assembler/disassembler function. The assembler/disassembler is an interactive assembler/editor in which the source program is not saved. Each source line is translated into the proper MC68000 machine language code and is stored in memory on a line-by-line basis at the time of entry. In order to display an instruction, the machine code is disassembled and the instruction mnemonic and operands are displayed. All valid MC68000 instructions are translated. The mnemonic ILLEGAL, described in Appendix B of the MC68000 User's Manual, is not recognized by the educational computer assembler. Also, refer to paragraph 4.2.2.4 for restrictions on the use of the mnemonic CCR.

The educational board assembler is effectively a subset of the MC68000 Resident Structured Assembler. It has more limitations than the resident assembler, such as not allowing line numbers and labels; however, it is a powerful tool for creating, modifying, and debugging MC68000 code.

4.1.1 M68000 Assembly Language

The symbolic language used to code source programs for processing by the assembler is called M68000 assembly language. This language is a collection of mnemonics representing:

- Operations
  - MC68000 machine-instruction operation codes
  - Directive (pseudo-op)
- Operators
- Special symbols

4.1.1.1 Machine-Instruction Operation Codes. That part of the assembly language that provides mnemonic machine-instruction operation codes for the MC68000 machine instructions is described in the MC68000 16-Bit Microprocessor User's Manual, MC68000UM. The user should reference this manual.

4.1.1.2 Directives. The assembly language can contain mnemonic directives which specify auxiliary actions to be performed by the assembler. Directives are not always translated to machine language.

Assembler directives assist the programmer:

- In controlling the assembler output
- In defining data and symbols
- In allocating storage

The educational board assembler recognizes only one directive called define constant (DC.W). This directive is used to define data within the program. Refer to paragraph 4.2.4 for a description of this directive.

4.1.2 Comparison with MC68000 Resident Structured Assembler

There are several major differences between the MEX68KECB assembler and the MC68000 Resident Structured Assembler. The resident assembler is a two-pass assembler that processes an entire program as a unit, while the educational board assembler processes each line of a program as an individual unit. Due mainly to this basic functional difference, the capabilities of the TUTOR assembler are more restricted:

a. Label and line numbers are not used. - Labels are used to reference other lines and locations in a program. The one-line assembler has no knowledge of other program lines and, therefore, cannot make the required association between a label and the label definition located on a separate line.

b. Source lines are not saved. - In order to read back a program after it has been entered, the machine code is disassembled and then displayed as mnemonic and operands.

c. Limited error indication. - The one-line assembler will show a question mark (?) under the portion of the source statement where an error probably occurred, or will display the word "ERROR" or other short message. In contrast, the resident assembler generates specific error messages for over 60 different types of errors.

d. Only one directive (DC.W) is accepted.

e. No macro operation capability is included.

f. No conditional assembly is used.

g. Several symbols recognized by the resident assembler are not included in the MEX68KECB assembler character set. These symbols include !, >, and <. Two other symbols, * and /, each have multiple meanings to the resident assembler, depending on the context, but only one meaning to the MEX68KECB assembler. Finally, the ampersand character (&) specifies a decimal number when used with the ECB assembler (although numbers with no prefix are assumed to be decimal) while this symbol represents a logical AND function to the resident assembler. Paragraph 4.2.1.5 describes the MEX68KECB assembler character set.

Although functional differences exist between the two assemblers, the one-line assembler is a true subset of the resident assembler. The format and syntax used with the TUTOR assembler are acceptable to the resident assembler except as described in g. above.

4.2 SOURCE PROGRAM CODING

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. Each source statement occupies a line and must be either an executable instruction or a DC.W assembler directive. Each source statement follows a consistent source line format.

4.2.1 Source Line Format

Each source statement is a combination of operation and, as required, operand fields; line numbers, labels, and comments are not used. The general format is:

sp <operation field> sp [<operand field>]

The space (sp) must be the first character of each line. This is to be consistent with the resident assembler, which expects the first field of each line to be either a space or a label. Because the TUTOR assembler never allows a label, the first character must always be a space.

4.2.1.1 Operation Field. The operation field must follow at least one space (more can be used) and entries can consist of one of two categories:

a. Operation codes - which correspond to the MC68000 instruction set, or

b. Define constant directive - the DC.W is recognized to define a constant in a word location. This is the only directive recognized by the assembler.

The size of the data field affected by an instruction is determined by the data size code. Some instructions and directives can operate on more than one data size. For these operations, the data size code must be specified or a default size applicable to that instruction will be assumed. The size code need not be specified if only one data size is permitted by the operation. The data size code is specified by a period (.), appended to the operation field, and followed by B, W, or L, where:

B = Byte (8-bit data)
W = Word (the usual default size; 16-bit data).
L = Long word (32-bit data)

The data size code is not permitted, however, when the instruction or directive does not have a data size attribute.

Examples (legal):

LEA	2(A0),A1	Long word size is assumed (.B,.W not allowed); this instruction loads effective address of first operand into A1.
ADD.B	(A0),D0	This instruction adds the byte whose address is (A0) to lowest order byte in D0.
ADD	D1,D2	This instruction adds low order word of D1 to low order word of D2. (W is the default size code.)
ADD.L	A3,D3	This instruction adds entire 32-bit (long word) contents of A3 to D3.

Example (illegal):

SUBA.B	#5,A1	Illegal size specification (.B not allowed on SUBA). This instruction would have subtracted the value 5 from the low order byte of A1; byte operations on address registers are not allowed.

4.2.1.2 Operand Field. If present, the operand field follows the operation field and is separated from the operation field by at least one space. When two or more operand subfields appear within a statement, they must be separated by a comma. In an instruction like ' ADD D1,D2' the first subfield (D1) is generally applied to the second subfield (D2) and the results placed in the second subfield. Thus, the contents of D1 are added to the contents of D2 and the result is saved in register D2. In the instruction ' MOVE D1,D2' the first subfield (D1) is the sending field and the second subfield (D2) is the receiving field. In other words, for most two-operand instructions, the general format ' opcode source,destination' applies.

4.2.1.3 Disassembled Source Line. The disassembled source line may not look identical to the source line entered. The disassembler makes a decision on how to represent a numerical value based on how it interprets the number's use. If the number is determined to be an address or a "would-be" address, it is displayed in hexadecimal; everything else is decimal. For example,

```
MOVE.L   #$1234, $5678
```

disassembles to

```
005000     21FC000012345678     MOVE.L   #4660,$00005678
```

Also, for some instructions, there are two valid mnemonics for the same op code, or there is more than one assembly language equivalent. The disassembler may choose a form different from the one originally entered. As examples:

a. BRA is returned for BT
b. DBF is returned for DBRA

NOTE

> The assembler recognizes two forms of mnemonics for two branch instructions. The BT form (branch conditionally true) has the same op code as the BRA instruction. Also, DBRA (decrement and branch always) and DBF (never true, decrement, and branch) mnemonics are different forms for the same instruction. In each case, the assembler will accept both forms.

4.2.1.4 Mnemonics and Delimiters. The assembler recognizes all MC68000 instruction mnemonics except ILLEGAL. Numbers are recognized as both decimal and hexadecimal, with decimal the default case (note that this is reverse to the TUTOR commands):

a. Decimal - is a string of decimal digits (0-9) without a prefix (default) or preceded by an optional ampersand (&). Examples are:

```
1234
&1234
```

b. Hexadecimal - is a string of hexadecimal digits (0-9, A-F) preceded by a dollar sign ($). An example is:

```
$AFE5
```

One or more ASCII characters enclosed by apostrophes (') constitute an ASCII string. ASCII strings are left-justified and zero-filled (if necessary), whether stored or used as immediate operands. This left justification will be to a word boundary if one or two characters are specified, or to a long word boundary if the string contains more than two characters.

```
005000    5300                  DC.W     'S'
005002    223C41424344          MOVE.L   #'ABCD',D1
005008    3536                  DC.W     '56'
```

NOTE

The MC68000 has seventeen 32-bit registers (D0-D7, A0-A6, SSP, USP) in addition to a 32-bit program counter (24 bits available) and a 16-bit status register. Registers D0-D7 are used as data registers for byte, word, and long word operations. Registers A0-A6 and SSP and USP are used as software stack pointers and base address registers; they may also be used for word and long word data operations. All 17 registers may be used as index registers. Register A7 is a pseudo register, used as the system stack pointer corresponding to either SSP or USP, depending on the operating state.

The following register mnemonics are recognized by the assembler:

D0-D7 Data Registers

A0-A7 Address Registers

Address register seven represents the system stack pointer of the active system state.

USP User stack pointer. Used only in privileged instructions which are restricted to supervisory state.

CCR Condition code register (low 8 bits of SR)

SR Status register. All 16 bits may be modified in the supervisor state. Only low 8 bits (CCR) may be modified in user state.

PC Program Counter. Used only in forcing program counter-relative addressing

4.2.1.5 Character Set. The character set recognized by the MEX68KECB assembler is a subset of ASCII, and these are listed below:

a. The uppercase letters A through Z

b. The integers 0 through 9

c. Arithmetic operators: + -

d. Parentheses ( )

e. Characters used as special prefixes:

(pound sign) specifies the immediate form of addressing
$ (dollar sign) specifies a hexadecimal number
& (ampersand) specifies a decimal number
@ (commercial at sign) specifies an octal number
% (percent sign) specifies a binary number
' (apostrophe) specifies an ASCII literal character

f. Five separating characters:

Space
, (comma)
. (period)
/ (slash)
- (dash)

g. The character * (asterisk) indicates current location.

4.2.2 Instruction Summary

The following paragraphs summarize the types of MC68000 instructions, their variations, and addressing modes. The MC68000 User's Manual describes the MC68000 instructions and addressing modes in greater detail.

4.2.2.1 Arithmetic Operations. The MC68000 instruction set includes the operations of add, subtract, multiply, and divide. Add and subtract are available for all data operand sizes, including extended, and also for address operands.

Multiply and divide may be signed or unsigned. Operations on decimal data (BCD) include add, subtract, and negate. The general form is:

```
OPERATION.SIZE    SOURCE,DESTINATION
```

Example:

```
ADD.W   D1,D2     Adds low order word of D1 to low order word of D2.

SUB.B   #5,(A1)   Subtracts 5 from the byte whose address is contained in A1.
```

4.2.2.2 MOVE Instruction. The MOVE instruction is used to move data between registers and/or memory. These moves include register-to-register, memory-to-memory, memory-to-register, and register-to-memory transfers. The general form is:

```
MOVE.SIZE     SOURCE,DESTINATION
```

Examples:

MOVE	D1,D2	Moves low order word of D1 into low order word of D2.
MOVE.L	$02000,$03000	Moves long word addressed by $02000 into long word addressed by $03000.
MOVE.W	#'A',1000	Moves word with value of 'A'00 into byte addressed by &1000.
MOVE	$2000,A3	Moves word addressed by $2000 into low order word of A3.

The MOVEQ mode always specifies a 32-bit destination operand and is, therefore, used only for a MOVE.L operation. The source data is eight bits and is sign-extended to a 32-bit value.

4.2.2.3 Compare Instructions. The general formats of the compare and check instructions are:

```
CMP.SIZE      OPERAND1,OPERAND2
CHK           BOUNDS,REGISTER
```

where $operand_1$ is compared to $operand_2$ by the non-destructive subtraction of $operand_1$ from $operand_2$ without altering $operand_1$ or $operand_2$.

Condition codes resulting from the subtraction include: N set for negative result, Z set for zero result, V set for overflow, and C set for a generated borrow.

The CHK instruction will cause a system trap if the register contents are less than zero or greater than the value specified by "bounds".

Examples:

CMP.L	$2000,D1	Compares long word at location $2000 with contents of D1, setting condition codes accordingly.
CHK	(A0),D3	Compares word whose address is in A0 with lower order word of D3; if check fails (see the MC68000 User's Manual), a system trap is initiated.

4.2.2.4 Logical Operations. Logical operations include AND, OR, EXCLUSIVE OR, NOT, and two logical test operations. These functions may be done between registers, between registers and memory, or with immediate source operands. The general form is:

```
OPERATION.SIZE     SOURCE,DESTINATION
```

Example:

```
AND   D1,D2        Low order word of D2 receives logical 'and' of low
                   order words in D1 and D2.
```

The destination may also be the status register (SR). When in the user state, only the lower eight bits of the status register may be modified. The byte size extension must be used. Note, however, that the mnemonic CCR is not accepted by the assembler for logical operations. Instead, the mnemonic SR must be used with a size extension of .B. CCR is used only with the MOVE instruction.

EXAMPLE: ANDI.B #5,SR instead of ANDI.B #5,CCR

4.2.2.5 Shift Operations. Shift operations include arithmetic and logical shifts, as well as rotate and rotate with extend. All shift operations may be either fixed with the shift count in an immediate field or variable with the count in a register. Shifts in memory of a single bit position left or right may also be done. The general form is:

```
OPERATION.SIZE     COUNT,OPERAND
```

Example:

```
LSL.W    #5,D3     Performs a left, logical shift of low order word of D3
                   by 5 bits; .W is optional (default).

ASR      #1,(A2)   Performs a right, arithmetic shift of the word whose
                   address is contained in A2; since this is a memory
                   operand, the shift is only 1 bit.

ROXL.B   D3,D2     Performs a right rotation with extend bit of low order
                   byte of D2; shift count is contained in D3.
```

4.2.2.6 Bit Operations. Bit operations allow test and modify combinations for single bits in either an 8-bit operand for memory destinations or a 32-bit operand for data register destinations. The bit number may be fixed or variable. The general form is:

```
OPERATION      BITNO,OPERAND
```

Example:

```
BCLR   #3,$44(A3)  Tests bit number 3 in byte whose address is given by
                   address in A3 plus displacement of $44, sets or clears
                   the Z condition code, and clears the specified bit in
                   the destination.

BCHG   D1,D2       Tests a bit in D2, reflects its value in condition code
                   Z, and then changes value of that bit; bit number is
                   specified in D1.
```

4.2.2.7 Conditional Operations. Condition codes can be used to set and clear data bytes. The general form is:

```
OPERATION     LOCATION
```

Example:

```
SNE   (A5)+        If condition code 'NE' (not equal) is true, then set
                   byte whose address is in A5 to 1's; otherwise, set that
                   byte to Ø's; increment A5 by 1.
```

4.2.2.8 Branch Operations. Branch operations include a branch to subroutine, an unconditional branch, and 14 conditional branch instructions. The general form is:

```
OPERATION.EXTENT     LOCATION
```

Examples:

```
003058    61A6        BSR     $3000    Branch to subroutine at location $3000.

003FF0    670E        BEQ.S   $4000    Short branch to $4000, on condition "EQ".

003FF0    6600000E    BNE.L   $4000    Long branch to $4000, on condition "NE".

003FF0                BPL.S   $3000    Short branch not allowed; displacement >
                                       8 bits.
```

All conditional branch instructions are PC-relative addressing only, and may be either one- or two-word instructions. The corresponding displacement ranges are:

```
one-word     -128...+127 bytes          (8-bit displacement)

two-word     -32768...+32767 bytes      (16-bit displacement)
```

By default, the assembler will resolve all references, both relative and absolute, by using the shorter form of the effective address in the operand reference, if possible; otherwise, the longer form will be chosen. The user can force the long form of the instruction by using the .L suffix.

In a short branch instruction, the operand must not reference the statement which immediately follows it. This would result in a displacement value of 0, which is recognized by the assembler as an error condition.

4.2.2.9 Jump Operations. Jump operations include a jump to subroutine and an unconditional jump. The general form is:

```
OPERATION.EXTENT      LOCATION
```

Example:

JMP	4(A7)	Unconditional jump to the location 4 bytes beyond the address in A7.
JMP.L	$2000	Long (absolute) jump to the address $2000.
JSR	$3000	Jump to subroutine at address $3000.

Jumps may specify any control addressing mode as the destination location. All references will use the shorter absolute address format, if possible; otherwise, the longer format will be used. The default extent may be overridden on a single jump operation to a label by appending "S" or "L" as an extent code for the instruction.

4.2.2.10 DBcc Instruction. This instruction is a looping primitive of three parameters: condition, data register, and address. The instruction first tests the condition to determine if the termination condition for the loop has been met and, if so, no operation is performed. If the termination condition is not true, the data register is decremented by one. If the result is -1, execution continues with the next instruction. If the result is not equal to -1, execution continues at the indicated location. The address must be within 16-bit displacement. The general format of the instruction is:

```
DBcc      DATA REGISTER,ADDRESS
```

4.2.2.11 Load/Store Multiple. This instruction allows the loading and storing of multiple registers. Its general format is:

```
MOVEM.SIZE  REGISTERS,LOCATION (register to memory)
MOVEM.SIZE  LOCATION,REGISTERS (memory to register)
```

where size may be either W (default) or L.

The "registers" operand may assume any combination of the following:

```
R1/R3/R6, etc., means R1 and R3 and R6
R1-R3, etc., means R1 through R3
```

The order in which the registers are processed is independent of the order in which they are specified in the source line; rather, the order of register processing is fixed by the instruction format. See MOVEM instruction in Appendix B of the MC68000 User's Manual for further details.

NOTE

Registers discussed here include data registers zero through seven and address registers zero through seven but not the software offset registers (R0 through R7) used by TUTOR.

Examples:

MOVEM (A6)+,D1/D5/D7	Load registers D1, D5, and D7 from three consecutive (sign-extended) words in memory, the first of which is given by the address in A6; A6 is incremented by 2 after each transfer.
MOVEM.L A2-A6,-(A7)	Store registers A2 through A6 in 5 consecutive long words in memory; A7 is decremented by 4 (because of .L); A6 is stored at A7; A7 is decremented by 4; A5 is stored at A7, etc.
MOVEM (A7)+,A1-A3/D1-D3	Loads registers D1, D2, D3, A1, A2, A3 in order from the six consecutive (sign-extended) words in memory, starting with address in A7 and incrementing A7 by 2 at each step.
MOVEM.L A1/A2/A3,$2000	Store registers A1, A2, A3 in three consecutive long words starting with location $2000.

4.2.2.12 Load Effective Address. This instruction allows computation and loading of an effective address into an address register. The general format is:

```
LEA   OPERAND,REGISTER
```

Example:

```
LEA   (A2,D5),A1        Load A1 with effective address specified by
                        first operand; see later explanation of
                        addressing mode "address register indirect
                        with index" (paragraph 4.2.3.2).
```

4.2.2.13 Variants on Instruction Types

Certain instructions allow a "quick" form when immediate data within a restricted size range appears as an operand. It is necessary for the programmer to "force" such a form by appending a "Q" to the mnemonic op code (to indicate "quick") on instructions for which such a form exists. If the specified quick form does not exist, or if the immediate data does not conform to the size requirements of the abbreviated form, an error will be generated.

Some instructions also have "address" variant forms (which refer to address registers as destinations); these variants append an "A" to the instruction mnemonic (e.g., ADDA, CMPA). This variant will be chosen by the assembler without programmer specification, when appropriate to do so; the programmer need specify only the general instruction mnemonic. However, the programmer may "force" or specify such a variant form by appending the "A". If the specified variant does not exist or is not appropriate with the given operands, an error will be generated.

The CMP instruction also has a memory variant form (CMPM) in which both operands are a special class of memory references. The CMPM instruction requires postincrement addressing of both operands. The CMPM instruction will be selected by the assembler, or it may be specified by the programmer.

The variations -- A, Q, and M -- must conform to the following restrictions:

A Must specify an address register as a destination, and cannot specify a byte size code (.B).

Q Requires immediate operand be in a certain size range. MOVEQ also requires longword data size.

M Both operands must be postincrement addresses.

For example, the instruction

```
ADDQ   #9,D0        Attempts to add value 9 to D0
```

will cause an assembly error, because the immediate operand is not in the valid size range (1 through 8).

4.2.3 Addressing Modes

Effective address modes, combined with operation codes, define the particular function to be performed by a given instruction. Effective addressing and data organization are described in detail in Section 2, "Data Organization and Addressing Capabilities", of the MC68000 User's Manual.

References to data addresses may be odd only if a byte is referenced. Data references involving words or long words must be even. Likewise, instructions must begin on an even word boundary.

Individual bits within a byte (operand for memory destinations) or long word (operand for data register destinations) may be addressed with the bit manipulation instructions (paragraph 4.2.2.6). Bits for a byte are numbered 7 to 0, with 7 being the most significant bit position and 0 the least significant. Bits for a long word are numbered from 31 to 0, with 31 being the most significant bit position and 0 the least significant bit position.

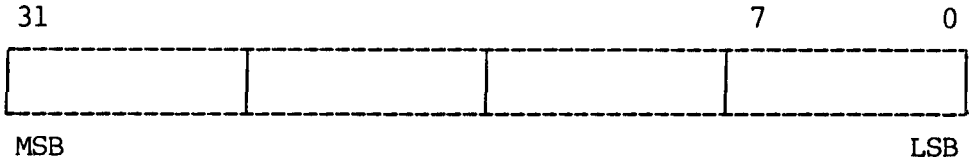

Table 4-1 summarizes the addressing modes defined for the MC68000, their syntax, and significant constraints.

TABLE 4-1. Address Modes

MODE	SYNTAX	COMMENTS
1) Register Direct		
a) Data register direct	Dn	
b) Address register direct	An	
2) Memory Address		
a) Address register indirect	(An)	
b) Address register indirect with predecrement	-(An)	
c) Address register indirect with postincrement	(An)+	
d) Address register indirect with displacement	<expr>(An)	<expr> must be absolute; displacement: 16 bits, sign-extended; register size is 32 bits (no options)
e) Address register indirect with index	<expr>(An,Am) <expr>(An,Dm)	<expr> must be absolute; displacement: 8 bits, sign-extended; note that A or D register may be used as index register (16 bits, sign-extended unless .L used); address register: 32 bits

TABLE 4-1. Address Modes (cont'd)

MODE	SYNTAX	COMMENTS
3) Special Address		
a) Absolute	<expr>	<expr> must specify an absolute address; there are two formats: absolute short is 16 bits, sign-extended; absolute long is 32 bits
b) PC with displacement	<expr>(PC)	forced PC with displacement; <expr> must be absolute; displacement: 16 bits, sign-extended
	<expr>(PC,An) <expr>(PC,Dn)	forced PC with index and displacement; <expr> must be absolute; displacement: 8 bits, sign-extended
c) Immediate data	#<expr>	<expr> must be absolute; may be used with .B, .W, .L
4) Implicit PC reference	—	Invoked by conditional branch (Bcc) or DBcc instruction; the effective address is a displacement from the PC; the displacement is either 8 or 16 bits, depending on the destination address and whether the default size is overridden on the current instructions.
5) Other implicit references	—	Some instructions make implicit reference to the system stack pointer (SP), the supervisor stack pointer (SSP), the user stack pointer (USP), or the status register (SR).

4.2.3.1 Register Direct Modes. These effective addressing modes specify that the operand is in one of the 17 multifunction registers (eight data and nine address registers). The operation is performed directly on the actual contents of the register. When an instruction is executed, references to A7 specify the supervisor stack pointer if the supervisor state status bit is set in the status register and the user stack pointer otherwise.

Notations:	An Dn	where n is between 0 and 7	Address register direct Data register direct

Examples:	CLR.L	D1	Clear all 32 bits of D1
	ADD	A1,A2	Add low order word of A1 to low order word of A2

4.2.3.2 Memory Address Modes. The following effective addressing modes specify that the operand is in memory and provide the specific address of the operand.

Address Register Indirect

The address of the operand is in the address register specified by the register field.

Notation: (An)

Examples:	MOVE	#5,(A5)	Move the value 5 to word whose address is contained in A5.
	SUB.L	(A1),D0	Subtract the value in the long word whose address is contained in A1 from D0.

Address Register Indirect with Predecrement

The address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four, depending upon whether the operand size is byte (.B), word (.W), or long (.L).

Notation: -(An)

Examples:	CLR	-(A2)	Subtract 2 from A2; clear word whose address is now in A2.
	CMP.L	-(A0),D0	Subtract 4 from A0; compare long word whose address is now in A0 with contents of D0.

Address Register Indirect with Postincrement

The address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four, depending upon whether the size of the operand is byte (.B), word (.W), or long (.L).

Notation: (An)+

Examples:	MOVE.B	(A2)+,D2	Move byte whose address is in A2 to D2; increment A2 by 1.
	MOVE.L	(A4)+,D3	Move long word whose address is in A4 to D3; increment A4 by 4.

Address Register Indirect with Displacement

The address of the operand is the sum of the address in the address register and the 16-bit sign-extended displacement.

Notation: <expression>(An)

Examples:	CLR.B	5(A0)	Clear byte whose address is given by adding 5 to contents of A0.
	MOVE	#2,10(A2)	Move 2 to word whose address is given by adding 10 to contents of A2.

Address Register Indirect with Index

The address of the operand is the sum of the address in the address register, the 8-bit sign-extended displacement, and the contents of the index (A or D) register.

Notations:	<expression>(An,Rn.W)	Specifies sign-extended low order word of index register.
	<expression>(An,Rn.L)	Specifies entire contents of index register.

Examples:	ADD	5(A1,D2),D5	Add to low order word of D5 the word whose address is given by addition of contents of A1, the sign-extended low order word of index register D2, and the displacement 5.
	MOVE.L	D5,$20(A2,A3.L)	Move entire contents of D5 to long word whose address is given by addition of contents of A2, contents of entire index register A3, and the displacement $20.

4.2.3.3 Special Address Modes. Special address modes use the effective address register field to specify the special addressing mode instead of a register number. The following table provides the ranges for absolute short and long addresses.

32-bit address	16-bit representation of 32-bit address	
00000000 . . 00007FFF	0000 . . 7FFF	Absolute Short
00008000 . . FFFF7FFF	No representation in 16 bits	Absolute Long
FFFF8000 . . FFFFFFFF	8000 . . FFFF	Absolute Short

Absolute Short Address

The 16-bit address of the operand is sign extended before it is used. Therefore, the useful address range is 0 through $7FFF and $FFFF8000 through $FFFFFFFF.

Notation: XXX

Example:

002500	4EF80400	JMP	$400	Jump to hex address 400 specified as a 16-bit sign-extended address.

Absolute Long Address

The address of the operand is the 32-bit value specified.

Notation: XXX

Examples:

007800	4EF900012000	JMP	$12000	Jump to hex address 12000 specified as a 32-bit address.
002500	4EF900000400	JMP.L	$400	Jump to hex address 400 specified as a 32-bit address. The long address is forced by the .L option.

Program Counter with Displacement

The address of the operand is the sum of the address in the program counter (current instruction location plus two) and the sign-extended 16-bit displacement integer. The assembler calculates this sign-extended displacement by subtracting the address of the displacement word (i.e., current instruction address plus two) from the value in the operand field.

Notation:	<expression>(PC)	Forced program counter-relative; cannot be used for branch instructions

The branch instructions (BRA, BSR, Bcc, DBcc) are a special case of the program counter with displacement address mode. These instructions always use program counter relative addressing; the displacement integer, however, can be either 16 or 8 bits long for the BRA, BSR, and Bcc instructions. An 8- or 16-bit displacement is specified by an .S or .L, respectively, following the instruction mnemonic. However, since these instructions allow only one address mode, the program counter with displacement mode is not explicitly selected in the source line. Instead, only the destination address of the branch is specified as shown in the following examples. For all other instructions, the program counter with displacement mode must be explicitly selected.

Examples:

001050	6700FFAE	BEQ.L	$1000	Branch if EQ condition code to $1000. Displacement integer is 16 bits.
001050	6E0E	BGT	*+$10	Branch if GT condition code to 16 bytes past this instruction.
001050	4EFAF8AE	JMP	$900(PC)	Force the evaluation of $900 to be program counter-relative. Displacement = $0900-$(1050+2)=$F8AE.

Program Counter with Index

The address is the sum of the address in the program counter, the sign-extended 8-bit displacement value, and the contents of the index (A or D) register. The displacement is calculated in the same manner as above.

Notations:	<expression>(PC,Rn.W) <expression>(PC,Rn.L)	Forced program counter-relative with index using word (default) or long word index.

Examples:

005000	4EFAC0FE	JMP	$1100(PC)	Force evaluation of $1100 to be program counter-relative. Displacement value is 16 bits.
005000		JMP	$1100(PC,A2)	Destination address is out of range; displacement value is only 8 bits for program counter with index address mode.
001150	4EFBA0AE	JMP	$1100(PC,A2)	Force evaluation of $1100 to be program counter-relative with index. Lower 16 bits of A2 are used as the index.
002030	323B58CE	MOVE	$2000(PC,D5.L),D1	Force evaluation of $2000 to be program counter-relative with index. All 32 bits of D5 are used as the index.

NOTE: In the program counter with displacement and program counter with index address modes, the expression represents the actual memory address. For example, to jump to address $1050, the instruction JMP $1050(PC) might be used. The assembler calculates the required displacement to reach address $1050 from the current location. In the address register indirect with displacement and the address register indirect with index address modes, however, the expression represents the displacement rather than the memory address -- hence, the instruction JMP $1050(A0) will jump to the memory address given by the contents of register A0 plus $1050.

Immediate Data

An absolute number may be specified as an operand by immediately preceding a number or expression with a '#' character. The immediate character (#) is used to designate an absolute number other than a displacement or an absolute address.

Notation: #XXX

Examples:	MOVE	#1,D0	Move value 1 to low order word of D0.
	SUB.L	#1,D0	Subtract value 1 from the entire contents of D0.

4.2.3.4 Notes on Addressing Options. By default, the assembler will resolve all references, both PC relative and absolute, by using the shorter form of the effective address in the operand reference, if possible; otherwise, the longer form will be chosen.

On an instruction which does not allow a size code, the reference default format may be overridden (for that instruction only) by appending .S (short) or .L (long) to the instruction mnemonic.

The shorter form of the effective address for relative branch instructions is an 8-bit displacement; the longer format is a 16-bit displacement. For absolute jumps, the shorter effective address is the 16-bit absolute short; the longer format is the 32-bit absolute long mode. In either relative branches or absolute jumps, if the shorter format is directed and the longer format is found necessary, an error will occur.

A long form may be forced by following the instruction mnemonic with .L

Example:

```
BEQ.L   $3050       If condition code 'EQ' (equal) is true, then branch to
                    $3050 (using the long form of the instruction).
```

In this case, the instruction size is forced to two words. An error will be printed if the operand field is not in the range of an 16-bit displacement.

Default actions of the assembler have been chosen to minimize two common address mode errors:

a. Displacement range violations

   Relative branch instructions (Bcc, BRA, BSR) allow either 8-bit or 16-bit displacements from the PC. On references in such instructions, the default action is to use the 8-bit displacement if the destination address is within that range; otherwise, the 16-bit displacement is used.

b. Inappropriate absolute short address

   Absolute addresses may be short (16-bit) or long (32-bit). On references with absolute effective address, the default action is to use the absolute short form if the address can be represented in 16 bits with sign extension; otherwise, the absolute long form is used.

4.2.4 DC.W Define Constant Directive

The format for the DC.W directive is:

```
sp    DC.W    <operand>
```

The function of the directive is to define a constant in memory. The DC.W directive can have only one operand (16-bit value) which can contain the actual value (decimal, hexadecimal, or ASCII). Alternatively, the operand can be an expression which can be assigned a numeric value by the assembler. The constant is aligned on a word boundary as word (.W) size is specified.

An ASCII string is recognized when characters are enclosed inside single quotes ('). Each character (7 bits) is assigned to a byte of memory, with the eighth bit (MSB) always equal to zero. If only one byte is entered, the byte is left justified. A maximum of two ASCII characters may be entered for each DC.W directive.

Examples are:

```
001022    04D2    DC.W    1234        Decimal number
001024    AAFE    DC.W    $AAFE       Hexadecimal number
001026    4142    DC.W    'AB'        ASCII string
001028    5443    DC.W    'TB'+1      Expression
00102A    4300    DC.W    'C'         ASCII character is left justified
```

4.3 ENTERING AND MODIFYING SOURCE PROGRAMS

User programs are entered into the Educational Computer RAM using the one-line assembler/disassembler. The program is entered in assembly language statements on a line-by-line base. The source code is not saved as it is converted immediately to machine code upon entry. This imposes several restrictions on the type of source line that can be entered.

Symbols and labels, other than the defined instruction mnemonics, are not allowed. The assembler has no means to store the associated values of the symbols and labels in lookup tables. This forces the programmer to use memory addresses and to enter data directly rather than use labels.

Also, editing is accomplished by retyping the entire new source line. Lines can be added or deleted by moving a block of memory data to free up or delete the appropriate number of locations.

In order to more clearly describe the procedures used to enter, modify, and execute a program, a specific example will be described. Figure 4-1 lists a program that converts an ASCII coded number into its hexadecimal equivalent. An ASCII character is in the lowest 8 bits of register D0 when the program is entered. Upon exiting, D0 contains the equivalent hexadecimal digit (0 to F), or an FF if the ASCII character does not correspond to a proper hex number.