

Introduction to Minicomputers

General Software



digital

1st Printing, June 1976
2nd Printing (Rev), October 1977
3rd Printing, August 1979

Copyright © 1976, 1977, 1979 by Digital Equipment Corporation

The reproduction of this workbook, in part or whole, is strictly prohibited. For copy information contact the Educational Services Department, Digital Equipment Corporation, Bedford, Massachusetts 01730.

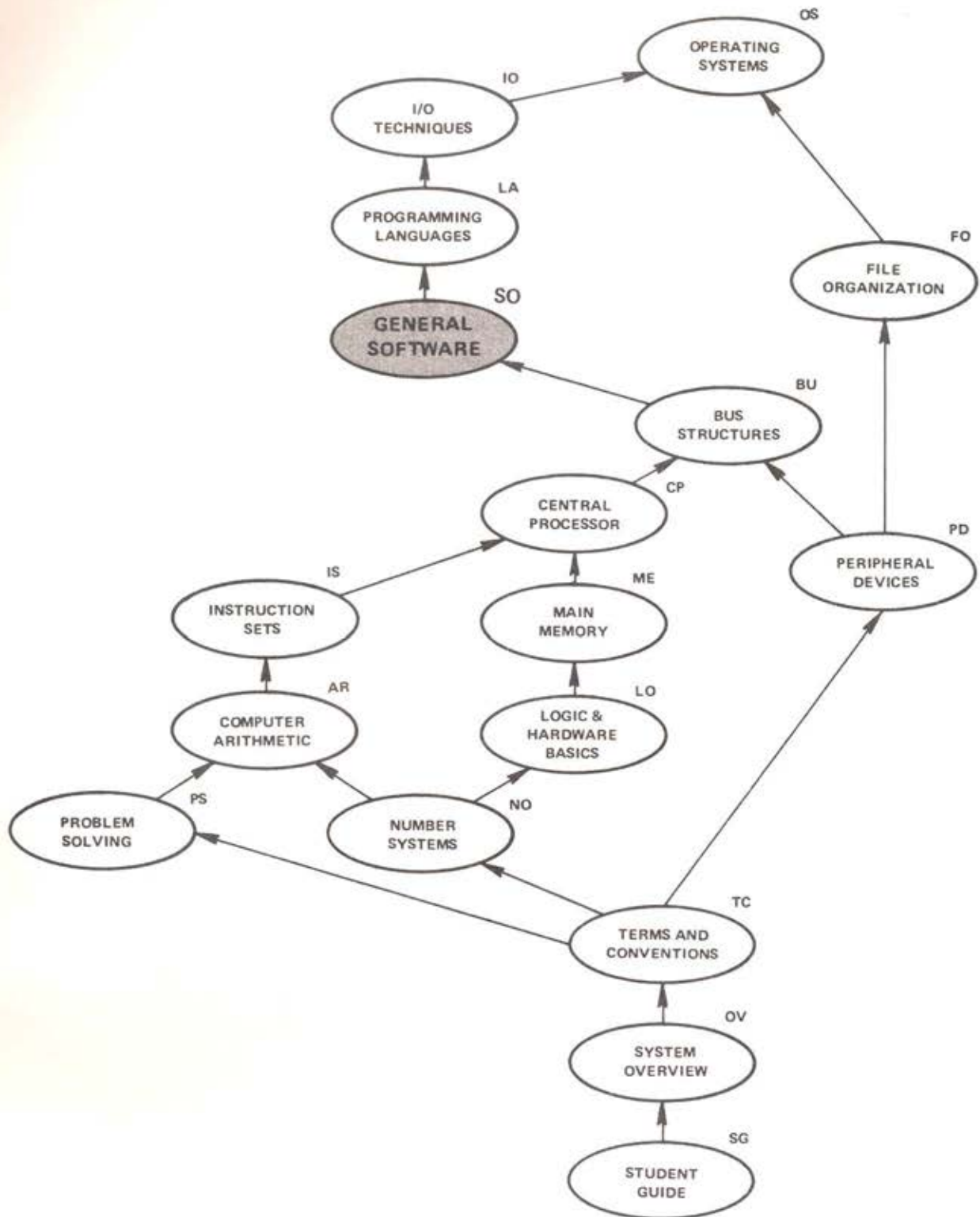
Printed in U.S.A.

INTRODUCTION TO MINICOMPUTERS

General Software

Student Workbook

COURSE MAP



CONTENTS

Introduction	1
Assemblers and Assembly Language	3
Objectives and Sample Test Items	3
Definitions	5
The Characteristics and Format of Assembly Languages	6
The Advantages of Programming in Assembly Language Instead of Machine Code	8
Exercises and Solutions	11
The Functions of an Assembler	15
Exercises and Solutions	21
High-Level Languages	33
Objectives and Sample Test Items	33
A Definition of a High-Level Language	35
Further Contrasts Between High- and Low-Level Programming Languages	36
Training Requirements for Programmers	37
Machine Independence	38
Ease of Documentation	38
Length of Development	39
Execution Time	39
Computer Resource Requirements for Translators	40
Exercises and Solutions	41

General Software

Introduction

Programs used by computers are called *software*, a term which provides a clear and useful distinction between computers themselves (the hardware) and programs that regulate computer activities and capabilities (the software).

As you know, computers are binary machines that can only execute instructions composed of patterns of 1s and 0s. Such instructions make up what is called the *machine code*. Because programming in machine code is both time consuming and error prone, nearly all computer software is generated by people who write, or *program*, in various computer languages.

A computer language is simply a set of special words and phrases that enables people to tell computers what to do. Each language also has special rules and punctuation symbols for combining these words into *instructions* or *statements*.

There are many different computer languages designed to facilitate the process of transforming an algorithm into a computer program. However, the choice of the language in which programmers implement an algorithm often determines the computer's efficiency in a given application.

The choice of the appropriate computer language is made easier by a number of distinctions that divide the wide spectrum of languages into several broad groups. This study unit focuses on the distinction between *low-level (assembly) languages* and *high-level languages* such as COBOL, FORTRAN, and BASIC.

This module is composed of two lessons. In Lesson 1, you will learn about low-level, or assembly, languages and their relationship to the binary machine code. You will also learn some of the functions of the *assembler*. An assembler is the program which *translates* assembly language into machine code. Lesson 2 introduces high-level languages, illustrates their advantages and disadvantages relative to assembly languages, and demonstrates the relationship of high-level languages to assembly languages.

Assemblers and Assembly Language

OBJECTIVES

1. Given five software concepts and five definitions, be able to match each concept with its definition.
2. Given seven descriptive statements, be able to select those statements that describe the advantages of programming in assembly language rather than machine code.
3. Given a table of eight assembler actions and three assembler passes, be able to match each action with the pass or passes in which it is executed.

SAMPLE TEST ITEMS

1. Match each of the following concepts with its definition by writing the correct letter in the space provided.

Concept	Definition
Source Program	_____
Symbol Table	_____
.	.
.	.
.	.

Definitions

- a. Form of a program as it is written by a programmer.
 - b. Establishes the relationships between labels and actual addresses.
 - .
 - .
 - .
2. Circle the letters of the statements that describe the advantages of using assembly language rather than machine code.
 - a. Faster to write.
 - b. Manipulates a greater number of binary addresses.
 - c. Less costly to use.
 - .
 - .
 - .

SAMPLE TEST ITEMS

3. For each of the actions listed below, place an X in the column corresponding to the assembler pass during which the action occurs.

NOTE

Some actions may occur in more than one pass. In such cases, indicate all passes in which the action may occur.

Action	Pass 1	Pass 2	Pass 3
Symbol Table			
a. The symbol table is constructed.			
b. The symbol table is printed.			
Binary Machine Code			
c. The binary machine code is generated.			
d. The binary code tape is punched.			
Assembly Listing			
e. The assembly listing is printed.			
Error Checking			
f. Duplicate labels are detected.			
g. Unresolved references are detected.			
h. Instruction syntax errors are detected.			

Before proceeding, you may wish to review the material covered in the study unit "Instruction Sets," paying particular attention to the format of machine-coded instructions. When you are ready, mark your place in this workbook and view Lesson I in the A/V program, "General Software."

Definitions

The following are definitions of the concepts covered in the audio-visual program.

Assembler	A program (usually supplied by the computer manufacturer) that translates the source program (in assembly language) into binary machine code.
Assembly	The process of translation from symbolic language (the source program) to binary machine code.
Assembly Language	The set of rules, symbols, and punctuation for the programmer to obey in writing symbolic language programs.
Assembly Language Program	A source program written according to the rules of an assembly language.
Assembly Listing	The side-by-side output of both the source program and the corresponding machine code generated by the assembler.
Binary Machine Code Program	A sequence of machine instructions with binary op (operation) codes and binary addresses. Sometimes the bit patterns are represented by octal notation.

Source Program	The form of the program as written by the programmer. The source program must be translated into machine code because the symbolic addresses and op codes used are not directly executable by a computer.
Symbol Table	An alphabetical ordering of symbols (labels) used in a source program. Each entry in the symbol table also contains the actual address associated with each individual label.

The Characteristics and Format of Assembly Languages

There are *four* important characteristics to remember about assembly languages:

1. The binary op codes of machine code are replaced by *mnemonic op codes* in assembly language. This eliminates having to remember bit patterns or octal numbers for operations and substitutes easy-to-remember words or abbreviations such as ADD for addition or STR for store.
2. The binary addresses of machine code are replaced by *symbolic addresses* in assembly language. Thus, the programmer does not have to calculate actual addresses but may label only those addresses to which he/she specifically wishes to refer.
3. Because of the mnemonic op codes and symbolic addresses, assembly language programs must be *translated* (or assembled) into machine code before they can be executed.
4. Each assembly language instruction corresponds to a *single machine code instruction*. Thus, the process of assembly is always a conversion to a single machine code instruction from each assembly language instruction.

A typical assembly language instruction contains up to four fields: *label, operation, operand, and comment fields*. In any instruction, all of these fields need not be used. The following is a sample assembly language instruction with four fields.

Label	Operation	Operand	Comment
START	ADD	NUMBER	/GET 1ST VALUE

The *label field* allows the labeled statement to be referenced symbolically to another place in the program. Any reference to START as an operand will refer to the location containing the ADD instruction. For example, suppose we wish to jump unconditionally to the ADD instruction. The following instruction accomplishes this:

JMP	START	/GO TO ADD INSTRUCTION
-----	-------	------------------------

Each occurrence of a label field causes a new entry to be inserted into the symbol table. This entry contains the new label and its associated addresses. If an identical label has already been assigned, then a conflict exists and an error message will be printed. Remember – *a label must always be unique within a program*. Good programmers also do not use more labels than necessary in order to minimize symbol table size and increase the efficiency of the assembler. Hence, only instructions that are referenced elsewhere should generally have label fields.

The *operation field* contains a mnemonic name of a computer operation contained in the binary machine code instruction set. Using the instruction set introduced in the study unit, "Instruction Sets," the mnemonic ADD is translated by the assembler into the binary op code of 0 0 1 (or octal code of 1). Thus, each op code in the binary machine code instruction set is represented by a mnemonic for the operation in the assembly language. In some assembly languages, the occurrence of a number (instead of a mnemonic) in the operation field indicates that the *numeric value* rather than an instruction is to be stored there. Thus,

NUMBER,	25	/DATA
---------	----	-------

instructs the assembler to place the numeric value 25 in the location labeled NUMBER.

The *operand field*, when present, contains the address of a memory location to be used with the instruction. Such an address is either the source or the destination of the action specified by the operation field. The name operand, therefore, denotes that *the address is being acted upon by the operation*. Therefore, our sample instruction

START ADD NUMBER /GET 1ST VALUE

means ADD the value, contained at the memory location labeled NUMBER, to the value in the accumulator. In this case, NUMBER is the source address of the value to be added. During the assembly of a program, the assembler inserts the actual address of the operand labeled NUMBER into the binary code produced.

The fourth field, the *comment field*, contains explanatory information concerning the instruction. Although the comment field is always optional, it should nearly always be used to help clarify the logic of the program. The contents of the comment field are totally ignored by the assembler and are used by *programmers* to document their program logic. The comment field in many assembly languages is frequently preceded by a special character such as / or ; To be useful, the comment should not merely repeat the mnemonic operation but explain *why* or *how* the operation is being done.

The Advantages of Programming in Assembly Language Instead of Machine Code

The advantages of assembly language programming over binary machine code programming are considerable and all result from mnemonic and symbolic qualities of the assembly language.

- *Assembly language is easier to learn.* The operation mnemonics are designed to require little or no memorization with regard to their meanings, and most of the memorization required involves remembering what operations are part of the instruction set. Certainly ADD as a mnemonic is easier to remember than 0 0 1 as an op code, although it will produce an addition operation in the computer.

- *Assembly language is faster to write.* The memory recall efficiency of the operation mnemonics is a factor here as well. In addition, the assembly language programmer is freed from the laborious process of calculating operand addresses by deferring address calculation to the assembler itself. As a result, a programmer can generate a program much quicker using assembly language.
- *Assembly language is less error prone.* Even an experienced programmer is likely to produce errors when programming in machine code for two reasons. First, the specification of the op codes and addresses must be precise, and there are many sources of confusion. Secondly, checking and editing machine code may be ineffective due to similarities between instructions. In assembly language, the differences are magnified. For example, STR A might be 011 010 000 111 in machine code and ADD A might be 001 010 000 111. While STR and ADD are quite different in assembly language, in the binary machine code there is only a difference in *one* bit between the instructions in the previous sentence.
- An assembly language program is *more easily documented*. Assembly language provides two important documentation aids which are not available in binary machine code. The use of labeled addresses allows data, as well as program steps, to have meaningful symbolic names. This feature combined with the valuable comment field allows an assembly language program to be nearly self-documenting. A binary machine code program must be documented externally and is nearly undecipherable without documentation.

These advantages are so significant that very few programs are even initially written in machine code. Also, the small cost of assembling the program into binary machine code is far overshadowed by the savings in programmer training, efficiency, and quality and by the usefulness of retaining programs for possible reuse by someone else at a later date.

EXERCISES

1. List three advantages of using symbolic language (**assembly language**) rather than machine code.
2. List the **four** fields possible in a typical assembly language instruction.
3. Differentiate between "source program" and "machine code."

SOLUTIONS

1. List three advantages of using symbolic language (**assembly language**) rather than machine code.
 - a. Easier to learn
 - b. Faster to write
 - c. Less error prone
 - d. More self-documenting
2. List the **four** fields possible in a typical assembly language instruction.
 - a. Label field
 - b. Operation field
 - c. Operand field
 - d. Comment field
3. Differentiate between "source program" and "machine code."

Source program: The form of the program as it is written by the programmer.

Machine code: A sequence of binary instructions that is executable by a computer.

EXERCISES

4. Define:

- a. Assembler
- b. Assembly
- c. Assembly language
- d. Assembly language program

SOLUTIONS

4. Define:

- a. **Assembler:** A program that converts a source program in assembly language into binary machine code.
- b. **Assembly:** The process of translating from a symbolic language to machine code.
- c. **Assembly language:** The set of rules for the programmer to obey in writing symbolic language programs.
- d. **Assembly language program:** A source program written according to the rules of an assembly language.

If you had trouble completing these exercises, view the A/V program again and/or reread the text. If certain points are still not clear, consult with your course manager.

The Functions of an Assembler

The assembler is a program usually supplied by the computer manufacturer. The assembler *translates* the source program of assembly language instructions into binary machine code capable of being executed by the computer.

The translation process is accomplished by multiple *passes* or *scans* through the source program. There are normally at least two passes required, and occasionally three or more passes are used. On small computers, each pass requires the operator to read in the source program and/or intermediate results. On larger machines, where there is sufficient memory for both the source program and the assembler to exist in memory *simultaneously*, the individual passes are invisible to the user. These multiple passes are required because not all the information necessary to produce a binary machine program can be gathered in a single pass.

The *first pass* (Figure 1) defines the symbolic addresses and the *second pass* uses them to generate code. The process of assembly is described in more detail, pass by pass, on the following pages.

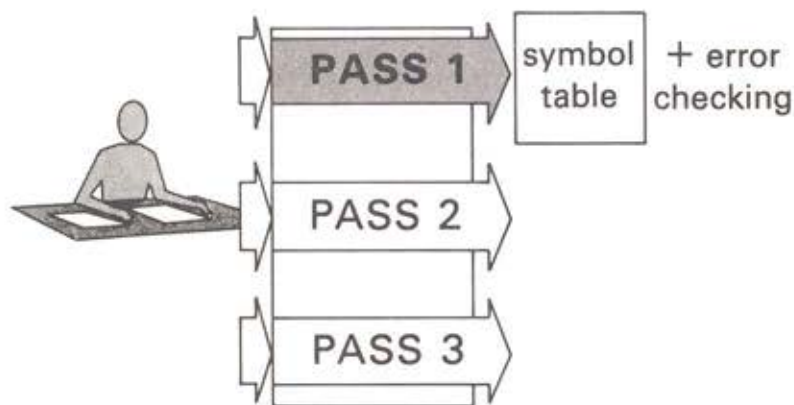


Figure 1 First Pass Operations

PASS 1:

- a. Symbol Table Creation
- b. Instruction Format Checking
- c. Duplicate Label Error Checking
- d. Print Out Symbol Table

The first pass of an assembler is concerned with **two** primary tasks:

1. Setting up a table equating symbolic addresses with actual addresses.
2. Checking that the instructions are properly constructed.

Each instruction is checked during the first pass to determine if it obeys the rules, or *syntax*, of the assembly language. If the instruction syntax is in error (caused by an improper or illegal operation mnemonic or an incorrectly formed operand), *assembly cannot proceed beyond the first pass*. All subsequent instructions will also be checked, but no code will be generated until the language syntax is correct in each instruction.

As the syntax of each instruction is being analyzed, the assembler also checks if there is a label field in the instruction. If there is a label, an attempt is made to insert a new entry for the label in the symbol table. Because a location counter is updated each time an instruction is processed, the assembler can also assign the associated actual address to the symbol. If an entry already exists for the "new" label, then the programmer has tried to define a symbol twice. This is an error which prohibits any subsequent passes from executing, because references to the symbol as an operand will be ambiguous – the assembler has no way to decide which of the multiple occurrences of the label is meant. Each such error will cause an error message to be printed, and the first pass will continue checking the remaining instructions before halting. At the end of the first pass, the symbol table and any error messages are printed out for the use of the programmer.

Thus, *a successful first pass produces a complete symbol table with the associated actual addresses*. An unsuccessful first pass occurs because one or more errors were detected, either in instruction syntax or in the duplicate label assignment. *An unsuccessful first pass prohibits execution of subsequent passes*.

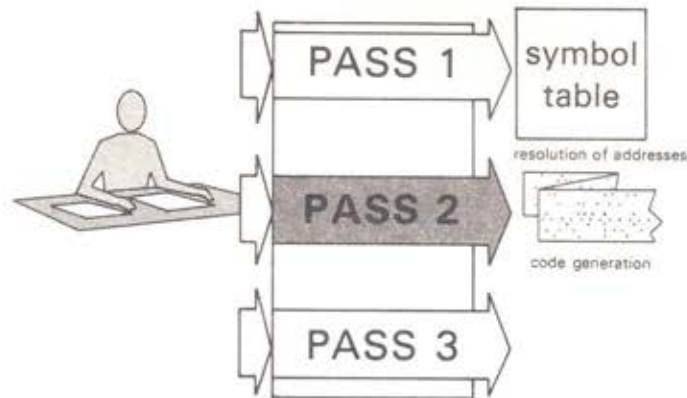


Figure 2 Second Pass Operations

PASS 2:

- a. Resolution of symbolic addresses
- b. Error checking for unresolved references
- c. Generation of the machine code

The second pass of an assembler (Figure 2) also has **two** primary tasks:

1. Resolution of symbolic addresses used in operands.
2. Generation of the machine code for the program.

Each instruction is now examined a second time. The instructions are processed individually and are *encoded into the binary machine code*. The operation mnemonic is converted into the binary op code, and the operand is converted into the binary operand (or address) code. However, many instructions cannot be directly encoded as they contain symbolic operands rather than actual addresses. Therefore, when an operand is symbolic, the assembler looks up the symbol in the symbol table, determines the associated actual address, and inserts the associated address as the binary operand address. *This process is called the resolution of an address*. However, an error can be detected at this stage that could not have been detected during the first pass – an *unresolved reference*. This error occurs when a symbolic operand uses a symbol that is never defined within the assembly language program, and is detected when the assembler attempts to look up the operand in the symbol table and cannot find an entry for it. An example should clarify this problem:

	Location	Instruction
	207	JMP NEXT
	210	_____
	211	_____
	212	_____
(NEXT?)	213	STR NUMBER

Let's assume that the programmer wished to jump from location 207 to 213 (the store instruction), but neglected to label the store instruction with the symbol NEXT. No entry would have been created in the first symbol table equating NEXT with location 213. When the second pass attempts to encode JMP NEXT, it is unable to assign a destination address for the jump.

An unresolved reference prohibits the second pass from generating correct machine code. Therefore, an error message is printed and code generation stops, although the assembler may continue detecting for further unresolved references. If no errors were detected during pass 2, then the output of the pass (frequently paper tape) is the translated program in machine code, ready for loading and execution.

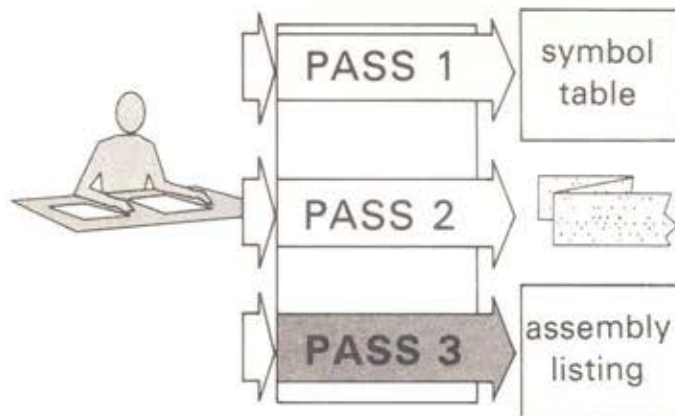


Figure 3 Third Pass Operations

PASS 3:

- a. Output of symbol table again
- b. Output of assembly listing

Many minicomputers have minimal input and output capabilities. I/O capabilities are frequently limited to a teleprinter, which can also read/punch paper tape. If both paper tape output of the machine code and a printed copy of the assembly listing are desired, then each must be done on a separate pass. The teleprinter may be either a printer or a punch at any moment, but it cannot print and punch simultaneously. Accordingly, the paper tape of the binary machine code is punched on the second pass, and all printing is deferred until the third (optional) pass (Figure 3). The symbol table and assembly listings are recommended under most circumstances to aid in debugging the program during the development phase and to assist in documenting the program when it is finished. This is why the third pass is nearly always used with minicomputers.

On computer systems with a line printer or a separate paper tape reader/punch, *simultaneous* printing of the symbol table and assembly listings and punching of the machine code paper tape is possible. There is, then, no third pass required.

The symbol table listing is simply a printed copy of the table showing the individual *equivalencies* of labels and addresses. This listing can be very useful after a program has stopped because of an error. In this case, the programmer can display on the console the contents of the address shown in the appropriate symbol table entry. This will reveal what the variable's value was at the time the program stopped.

The assembly listing is a side-by-side printing of both the generated machine code and the original assembly language source program. Also included is the location corresponding to each instruction of the program. The listing serves not only as a legible printed copy of both source and machine code versions of the program but also as a debugging tool during program development.

Most minicomputers provide the ability to step through a program manually one instruction at a time. In this way, a program's logic can be confirmed or an error detected. The assembly listing (Figure 4) provides a ready reference of the instructions and their locations, as well as a quick means for retranslating from the machine code back to the assembly language. Therefore, it is a useful tool for monitoring program behavior and correcting logical errors.

177570	SWR=177570				
					I CONSOLE SWITCH REGISTER
000000	. = 0				I RELOCATABLE
00000 010706	START: MOV	PC, SP			
00002 005746	TST	-(SP)			I SET UP STACK POINTER
00004 010600	MOV	SP, R0			I GET ABSOLUTE ADDRESS
00006 062700	ADD	#TTSERV-START, R0			I OF TRACE TRAP SERVICE
00008 000054					
00012 010037	MOV	R0, #14			I LOAD INTO TRACE TRAP VECTORS
00014 000014					
00016 012737	MOV	#340, #16			I PRIORITY 7 INHIBITS INTERRUPTS
00018 000340					
00020 000016					
00024 000000	HALT				I WAIT FOR OPERATOR
00026 013700	MOV	#SWR, R0			I GET SWR CONTENTS
00028 177570					
00032 042700	BIC	#160001, R0			I CONVERT TO REALISTIC ADDR
00034 160001					
00036 012746	MOV	#20, -(SP)			I SET T BIT FOR PROGRAM
00038 000020					
00042 010046	MOV	R0, -(SP)			I AND NEW PC FOR START ADDR
00044 010067	MOV	R0, FLAG			I (FLAG <-- START ADDR) ==> 1ST TIME
00046 000002					
00050 000002	RTI				I TRANSFER TO PROGRAM
00052 000000	FLAG: .WORD 0				I FLAG WORD

Figure 4 Sample Assembly Listing

You have now learned about the format and characteristics of assembly languages, the advantages of coding in assembly languages rather than coding directly in binary machine code, and the major functions and steps of an assembler. Do the following exercises before continuing to Lesson 2.

EXERCISES

1. List five functions of an assembler, giving two operations performed in Pass 1, two operations performed in Pass 2, and at least one operation performed in Pass 3.

Pass 1:

a.

b.

Pass 2:

a.

b.

Pass 3:

a.

b.

SOLUTIONS

1. List five functions of an assembler, giving two operations performed in Pass 1, two operations performed in Pass 2, and at least one operation performed in Pass 3.

Pass 1:

- a. Symbol table generation
- b. Syntax error checking
- c. Duplicate label error checking
- d. Output of symbol table

Pass 2:

- a. Unresolved reference error checking
- b. Output of the machine code

Pass 3:

- a. Output of the symbol table
- b. Output of the assembly listing

EXERCISES

2. Examine the following sample program in assembly language:

	CLA	/CLEAR ACCUMULATOR
START,	ADD NUMBER	/GET FIRST VALUE
	ADD NUMBER	/CONTINUE WITH
	ADD NUMBER	/ADDITIONAL
	ADD NUMBER	/VALUES TO BE ADDED
	STR SUM	/SAVE RESULT
	HLT	
NUMBER,	7	
SUM	0	

Equivalencies

Mnemonic	Machine Code
CLA	7200
ADD	1xxx
STR	3xxx
HLT	7402
"Data"	0yyy

where:

xxx = the octal address of the operand
yyy = the octal data value

Assume that the starting location of the program will be at 200 and that each instruction requires one memory location. Fill in the information for one pass of the assembler at a time. Check your answers before proceeding to the next pass. This will eliminate any possible cumulative errors.

(Continue Exercise 2 on page SO 25.)

EXERCISES

- 2a. Fill in all information which is known after the completion of the FIRST PASS. Also write the word "OUTPUT" under any information which is printed or punched during the pass.

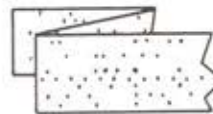
Source Program			Mnemonic	Machine Code
START,	CLA		CLA	7200
	ADD	NUMBER	ADD	1xxx
	ADD	NUMBER	STR	3xxx
	ADD	NUMBER	HLT	7402
	STR	SUM	"Data"	0yyy
	HLT			
NUMBER,	7			
SUM,	0			

PASS 1:

Symbol Table

Symbol	Address

Binary Machine Code Tape



Assembly Listing

LOC	Machine Code	Label	Mnemonic	Operand
200				
201				
202				
203				
204				
205				
206				
207				
210				

SOLUTIONS

- 2a. Fill in all information which is known after the completion of the FIRST PASS. Also write the word "OUTPUT" under any information which is printed or punched during the pass.

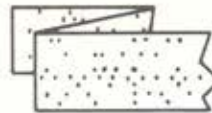
Source Program

200		CLA	
201	START,	ADD	NUMBER
202		ADD	NUMBER
203		ADD	NUMBER
204		ADD	NUMBER
205		STR	SUM
206		HLT	
207	NUMBER,	7	
210	SUM,	0	

Symbol Table

Symbol	Address
NUMBER	207
START	201
SUM	210

Binary Machine Code Tape



OUTPUT

Assembly Listing

LOC	Machine Code	Label	Mnemonic	Operand
200				
201				
202				
203				
204				
205				
206				
207				
210				

REMEMBER – The first pass orders the symbol table alphabetically. The associated address is the location of the labeled instruction. The symbol table is output at the end of pass 1.

EXERCISES

- 2b. Fill in all information which is known after the completion of the SECOND PASS. Also write the word "OUTPUT" under any information which is printed or punched during the pass.

Source Program			Mnemonic	Machine Code
START,	CLA			
	ADD	NUMBER		
	ADD	NUMBER		
	ADD	NUMBER		
	ADD	NUMBER		
	STR	SUM		
	HLT			
NUMBER,	7			
SUM,	0			

Mnemonic	Machine Code
CLA	7200
ADD	1xxx
STR	3xxx
HLT	7402
"Data"	0yyy

PASS 2:

Symbol Table

Symbol	Address
NUMBER	207
START	201
SUM	210

Binary Machine Code Tape



Assembly Listing

LOC	Machine Code	Label	Mnemonic	Operand
200				
201				
202				
203				
204				
205				
206				
207				
210				

SOLUTIONS

- 2b. Fill in all information which is known after the completion of the SECOND PASS. Also write the word "OUTPUT" under any information which is printed or punched during the pass.

Source Program

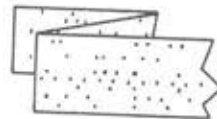
	CLA	
START,	ADD	NUMBER
	ADD	NUMBER
	ADD	NUMBER
	ADD	NUMBER
	STR	SUM
	HLT	
NUMBER,	7	
SUM,	0	

PASS 2:

Symbol Table

Symbol	Address
NUMBER	207
START	201
SUM	210

Binary Machine Code Tape



7200
1207
1207
1207
1207
1207
3210
7402
0007
0000
OUTPUT

Assembly Listing

LOC	Machine Code	Label	Mnemonic	Operand
200				
201				
202				
203				
204				
205				
206				
207				
210				

The second pass produces the machine code for execution.

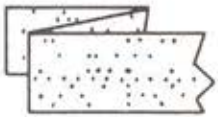
EXERCISES

- 2c. Fill in all information which is known after the completion of the THIRD PASS. Also write the word "OUTPUT" under any information which is printed or punched during the pass.

Source Program			Mnemonic	Machine Code
START,	CLA		CLA	7200
	ADD	NUMBER	ADD	1xxx
	ADD	NUMBER	STR	3xxx
	ADD	NUMBER	HLT	7402
	STR	SUM	"Data"	0yyy
	HLT			
NUMBER,	7			
SUM,	0			

PASS 3:

Symbol Table	
Symbol	Address
NUMBER	207
START	201
SUM	210

Binary Machine Code Tape	
	7200
	1207
	1207
	1207
	1207
	3210
	7402
	0007
	0000

Assembly Listing

LOC	Machine Code	Label	Mnemonic	Operand
200				
201				
202				
203				
204				
205				
206				
207				
210				

SOLUTIONS

- 2c. Fill in all information which is known after the completion of the THIRD PASS. Also write the word "OUTPUT" under any information which is printed or punched during the pass.

Source Program

```

START,    CLA
          ADD    NUMBER
          ADD    NUMBER
          ADD    NUMBER
          ADD    NUMBER
          STR    SUM
          HLT
NUMBER,   7
SUM,      0
  
```

PASS 3:

Symbol Table

Symbol	Address
NUMBER	207
START	201
SUM	210

OUTPUT

Binary Machine Code Tape



7200
 1207
 1207
 1207
 1207
 3210
 7402
 0007
 0000
 OUTPUT

Assembly Listing

LOC	Machine Code	Label	Mnemonic	Operand
200	7200		CLA	
201	1207	START	ADD	NUMBER
202	1207		ADD	NUMBER
203	1207		ADD	NUMBER
204	1207		ADD	NUMBER
205	3210		STR	SUM
206	7402		HLT	
207	0007	NUMBER,	7	
210	0000	SUM,	0	

OUTPUT

REMEMBER: During the optional third pass, both the symbol table and the assembly listing may be printed.

If you had trouble completing these exercises, view the A/V program again and/or reread the text. If certain points are still not clear, consult with your course manager.

High-Level Languages

OBJECTIVES

1. Given statements referring to high-level programming languages, be able to label those statements that refer to advantages of using high-level rather than low-level programming languages, those that refer to disadvantages of using high-level programming languages, and those that refer to neither advantages nor disadvantages.
2. Given a mathematical expression written in a high-level language, a list of equivalent assembly instructions, and a list of storage locations and contents, be able to write the steps of the simple program necessary to convert the mathematical expression into equivalent assembly language instructions.

SAMPLE TEST ITEMS

1. Indicate that each of these statements is an advantage (A) of using high-level programming languages, a disadvantage (D), or neither (N) an advantage nor disadvantage by writing the correct letter in the space provided.

Statement	A, D, or N
Uses a different amount of execution time and memory space than low-level languages.	_____
Efficiency of a high-level language translator differs from that of a low-level language translator.	_____
Similarity to natural language and algebra.	_____
.	.
.	.
.	.

SAMPLE TEST ITEMS

2. Using the instruction set below, convert the statement:

$$\text{RESULT} = A + B - C + D$$

into equivalent assembly instructions.

CLA
ADD
STR
CMA
IAC
"Data"

- As the statement would most likely be part of a larger program, a *halt* statement would *not* be inserted after the operation is coded.
- It is *not* necessary to convert the instructions into binary machine code.
- RESULT, A, B, C, and D are stored at successive locations beginning at location 205. Assume that the instructions for the arithmetic are to begin at location 317.

$$\text{RESULT} = A + B - C + D$$

Location	Label	Operand
205	RESULT,	0
206	A,	1
207	B,	7
210	C,	5
211	D,	10

317

Mark your place in this workbook and view Lesson 2 of the audio-visual program, "General Software."

A Definition of a High-Level Language

All programming languages have certain common characteristics:

- They are designed for expressing procedures.
- They have their own vocabularies, grammar, and punctuation values as natural languages do.
- They must be translated into machine code before a computer can execute programs written in them.

The previous lesson discussed assembly languages, which are examples of low-level languages for two reasons:

- Assembly language instructions have a one-to-one correspondence with elementary computer operations.
- An individual instruction, despite the use of symbolic addresses and operation mnemonics, bears a closer resemblance to the machine code than it does to algebraic or natural language expression.

High-level programming languages, of which there are many, are distinctly different from low-level languages:

- High-level language *statements* (or *lines*) have a many-to-one correspondence with both machine code and assembly language instructions. Many of the elementary computer operations and hardware considerations are removed from the programmer's concern and control.
- High-level language statements generally correspond closely to natural language or algebraic expressions encountered in everyday life.

Thus, for example,

LET $X = A + B - C$

is more understandable and more easily learned than

CLA	/CLEAR ACCUMULATOR
ADD C	/LOAD VALUE OF C INTO AC
CMA	/AC = -C (one's complement)
IAC	/AC = -C (two's complement)
ADD B	/AC = B - C
ADD A	/AC = A + B - C
STR X	/STORE RESULT

which accomplishes the same result.

Thus, high-level languages generally are tailored to favor the people who must use them, rather than the computers on which the programs will be run.

Further Contrasts Between High- and Low-Level Programming Languages

Criterion	High Level	Low Level
1. Programmer training required for:		
a. Common algorithms	Yes	Yes
b. Language rules	Less	More
c. Hardware functions	No	Yes
2. Machine independence	Yes	No
3. Ease of documentation	Yes	No
4. Development time	Shorter	Longer
5. Execution time	Longer	Shorter
6. Computer resources required for translation (time and memory size)	More	Less

Training Requirements for Programmers

Computer programmers must have training in three areas:

1. Common algorithms
2. Rules for a language
3. Hardware functions

Regardless of the language to be used, programmers must possess knowledge of certain common algorithms. (A good example is how to sort items efficiently.) This body of common algorithms is independent of the languages; thus, the requirements are the same for both high- and low-level languages.

A programmer must also know the rules for a language in order to use it correctly. When the language is a high-level one, the time required for a programmer to learn it is much shorter than it would be for a low-level language. This situation is caused by the fact that the high-level language statements compare closely with the natural language and algebraic backgrounds of the programmer. Low-level languages cannot benefit to any large degree from programmer experience in everyday life. For this reason, training time is increased.

The third major area of necessary expertise is concerned with hardware functions. When programmers write source programs in a high-level language, they are freed from most hardware considerations – the translator does this part of a programmer's job. A low-level language, however, does not provide this assistance: a seemingly simple process such as printing a number can be a reasonably complex process. Therefore, training programmers in hardware functions is necessary for the use of low-level languages, but the use of high-level languages minimizes this training.

Machine Independence

Low-level languages possess a one-to-one correspondence to elementary machine operations. They must, therefore, be intimately dependent on a specific instruction set. Because instruction sets tend to be unique for each family of computers, low-level language programs are seldom shared among users with different computers.

High-level languages do possess some measure of machine independence. The most popular high-level languages are standardized either by long-standing use and support or by actual imposition of standards by an official agency. Thus, a program written in a popular high-level language may be shared among installations with widely varying computers. Although each installation may need to make minor changes to suit a specific computer, the bulk of the program can be left untouched. High-level languages are preferable if programs are to be distributed to other installations or computers.

Ease of Documentation

Natural language and algebraic-looking statements do much to make high-level languages self-documenting. When the programmer also uses mnemonic variable names and makes judicious use of comments, a program can become truly self-documenting.

Low-level languages absolutely require comments if there is to be any hope of adequate documentation. Frequently, however, the number of instructions required to perform a single task may become large enough that the important logical step may be obscured by all the minor detailed instruction logic. Even the authors of an assembly language program may take quite a while to understand their own program if they have not seen it in six months or a year.

Consequently, high-level language programs are more likely to be understood by others than low-level language programs.

Length of Development

Two features cause the program development time to be shorter for high-level languages than for low-level languages. First, the similarity of high-level languages to the programmer's everyday experience enables them to express their ideas more freely when writing the source program. The programmer has to perform less translation of terms and logic when working in a high-level language.

Secondly, surveys indicate that programmers tend to average approximately 20 lines of debugged, documented code each day – regardless of the language used. While 20 lines of low-level language translates into 20 instructions of machine code, 20 lines of high-level language programming may yield several hundred machine code instructions.

Thus, on a daily basis and using the resultant machine code instruction counts as the measure, a programmer may easily be ten to twenty times more productive when using high-level languages. In light of recent trends of rapidly increasing software cost and decreasing hardware costs, the dramatically reduced costs of development present an important argument for the use of high-level languages.

Execution Time

Low-level languages enable a good programmer to squeeze the utmost efficiency from a program, because the programmer can control every operation at its most basic level. Such techniques as overlapping competing input/output operations and minimizing memory references can sometimes significantly increase execution speed.

The high-level language programmer, however, has sacrificed much of the ability to control such basic operations in return for being freed from most hardware considerations. Most high-level language translators, therefore, make an effort to optimize the translated code they produce.

In general, however, better-than-average, low-level programmers will produce more efficient machine code (through an assembler) than will high-level language programmers (through the language translator). Hence, in applications where execution speed is critical, low-level languages have an advantage.

Computer Resource Requirements for Translators

Low-level languages are more efficient in the use of both time and storage during the translation of source programs. Because the process of translation is much simpler and on a one-to-one basis, assemblers occupy much less memory than do high-level language translators. Secondly, the simpler translation process enables low-level languages to be translated much faster than high-level languages.

As we shall see in the next study unit, "Programming Languages," many high-level language translators first translate from high-level language to low-level language, and then retranslate the program into machine code. The translation process is then obviously more expensive in terms of time and memory for a high-level language.

You have learned several distinctions between high- and low-level languages as well as some advantages and disadvantages of each. You should also be able to convert a simple statement such as

$$X = A + B - C$$

into equivalent assembly language such as

```
CLA
ADD C
CMP
IAC
ADD B
ADD A
STR X
```

Starting on the next page are exercises that should re-emphasize what you have learned in this lesson. When you feel you understand both Lesson 1 and Lesson 2, take the module test.

EXERCISES

1. Using the instruction set given below, convert $X = A + B$ into equivalent assembly language instructions.

Available instruction set:

ADD M	Means add value at location M to value in accumulator.
STR Z	Means store value of accumulator (AC) into location Z.
CLA	Means clear accumulator.
CMA	Means take complement of AC.
IAC	Means increment the AC by 1.

2. Using the instruction set given in Exercise 1, convert:

$$X = A - B$$

SOLUTIONS

1. Using the instruction set given below, convert $X = A + B$ into equivalent assembly language instructions.

Available instruction set:

ADD M	Means add value at location M to value in accumulator.
STR Z	Means store value of accumulator (AC) into location Z.
CLA	Means clear accumulator.
CMA	Means take complement of AC.
IAC	Means increment the AC by 1.

CLA	/CLEAR AC
ADD B	/GET FIRST VALUE
ADD A	/ADD 2ND VALUE
STR X	/X = A + B

2. Using the instruction set given in Exercise 1, convert:

$$X = A - B$$

CLA	/CLEAR AC
ADD B	/GET FIRST VALUE = B
CMA	/COMPLEMENT = -B (one's complement)
IAC	/INCREMENT AC = -B (two's complement)
ADD A	/ADD SECOND VALUE = A - B
STR X	/X = A - B

EXERCISES

3. Using the instruction set given below, convert

$$X = A - B + C - D - E$$

into the equivalent assembly language instructions.

Available instruction set:

ADD M Means add value at location M to value in accumulator

STR Z Means store value of accumulator into location Z

CLA Means clear accumulator

CMA Means take complement of AC (that is, change the sign of the number in the AC).

IAC Means increment the AC by 1

Hint: $X = A - B + C - D - E$ is equivalent to

$$X = A + C - (B + D + E)$$

SOLUTIONS

3. Using the instruction set given below, convert

$$X = A - B + C - D - E$$

into the equivalent assembly language instructions.

Available instruction set:

ADD M Means add value at location M to value in accumulator

STR Z Means store value of accumulator into location Z

CLA Means clear accumulator

CMA Means take complement of AC (that is, change the sign of the number in the AC).

IAC Means increment the AC by 1

Hint: $X = A - B + C - D - E$ is equivalent to

$$X = A + C - (B + D + E)$$

CLA /CLEAR AC FOR ARITHMETIC

ADD E /LOAD 1ST VALUE E

ADD D /ADD NEXT VALUE D + E

ADD B /SUBTRACTION TERMS NOW COMBINED B + D + E

CMA /TAKE COMPLEMENT FOR SUBTRACTION - (B + D + E)

IAC /INCREMENT AC TO TWO'S COMPLEMENT

ADD C /ADD NEXT VALUE C - (B + D + E)

ADD A /EXPRESSION COMPLETE A + C - (B + D + E)

STR X /STORE RESULT X = A + C - (B + D + E)

This represents an optimal solution. There are many other possible solutions – all of which require more memory references and instructions. These other solutions may not necessarily be wrong. However, they are less efficient.

EXERCISES

4. Fill in the following chart listing the contrasts between high- and low-level languages. Use word pairs such as

yes – no
less – more
larger – smaller
faster – slower
longer – shorter

to describe the contrasts.

Criterion	High-Level	Low-Level
Programmer training required		
Machine independence		
Ease of documentation		
Development time		
Execution time		
Required computer resources for translation		

SOLUTIONS

4. Fill in the following chart listing the contrasts between high- and low-level languages. Use word pairs such as

yes – no
less – more
larger – smaller
faster – slower
longer – shorter

to describe the contrasts.

Criterion	High-Level	Low-Level
Programmer training required	Less	More
Machine independence	More or yes	Less or no
Ease of documentation	More or yes	Less or no
Development time	Shorter	Longer
Execution time	Longer	Shorter
Required computer resources for translation	More or larger	Less or smaller

If you had trouble completing these exercises, view the A/V program again and/or reread the text. If certain points are still not clear, consult with your course manager.

Take the test for this module and evaluate your answers before studying another module.