Systems Software

A journey of a thousand miles begins with a single step.
—Lao Tzu
Tao Te Ching, chapter 64

INTRODUCTION

When someone wants to solve a problem using a computer, an algorithm to solve it is required, and that algorithm must be expressed in a programming language. In general, people think programming languages were created to instruct the computer to carry out a specific task or application. This is partially true, because programming languages were created mainly, as Donald Knuth says in his work Literate Programming, "to explain to human beings what we want the computer to do." A series of transformations is needed to convert the program written by the programmer into its executable version, and all those transformations are carried out using different programs classified as systems software. When the executable version is ready for execution, once again, systems software programs are required to create the environment where the executable version will run. Systems software programs will interact with programmers from the moment the program is first written until the moment the executable version of the program runs to completion.

In this chapter we show the landscape of what this book is about. We initiate the chapter by defining systems software. Then we present a classification of systems software programs, basically in two groups. The first group encompasses programs to help programmers develop

CHAPTER OBJECTIVES

- To introduce the concept of systems software.
- To understand the difference between systems software programs used for program development and systems software programs required for program execution.
- To explore all transformations a program passes through from the moment it is written to the moment the program is executed.

applications, and the second set includes programs that create an environment that allow the execution of application programs. We show all steps required to transform a progra written in high-level language into a program in execution or process, and for each step, w identify which one of the systems software programs is used. Finally, we present an outlin of the content of each chapter.

VOCABULARY

This is a list of keywords that herald the concepts we will study in this chapter. They are orga nized in chronological order and will appear in the text in bold. We invite you to take a look a these keywords to find out which ones you are familiar with.

Systems software
Text editors
Compilers
Assemblers
Linker
Operating systems
Loaders
Dynamic linkers
Program libraries
Executable program
Assembly language
Electronic Delay Storage Automatic Calculator (EDSAC)
Fortran
Screen editors
Object code
Linking
Text section
Data section
Relocation section
Symbol table
Executable and Linkable Format (ELF)
Process
Runtime environment
Command interpreter
Process address space
Text
Data
Heap
Stack
Process control block (PCB)
Process ID
Program counter (PC)
Stack pointer (SP)
State
CPU registers
Dispatching

ACTIVATING PRIOR KNOWLEDGE

In this section we will present a series of activities. In some of them you can choose one or more options. Sometimes, if you do not agree with the given answers to choose from, you will be allowed to give your own answer. By the way, this is not a test.

1. Which one of these programs can be classified as systems software?

   Text editor ☐
   A program to compute factorial ☐
   Compiler ☐
   Loader ☐

2. Mark the words denoting the same concept.

   Assembler ☐ Compiler ☐ Translator ☐

3. Mark the programs you have used.

   Compiler ☐ Text editor ☐ Command interpreter ☐

4. Is it true that Program = Process?

   Yes ☐
   No ☐
   Sometimes ☐

5. Skim the chapter and pay attention to the words written in **bold**. Count the number of words you are familiar with.

6. To execute a program, the Operating System (OS) must load the program from hard disk. Where does the operating system load programs in the computer?

   In the CPU ☐
   In memory ☐
   50 percent in memory and 50 percent in the CPU ☐

7. Have you used any of the following programs?

   Linker ☐
   Loader ☐
   Debugger ☐
   Assembler ☐
   Text editor ☐

What Is Systems Software?

When you ask someone what systems software is, there is a tendency to answer by naming one or two of the programs considered as systems software. Therefore, we will start out with Leland L. Beck's definition of systems software:

> **Systems software** consists of a set of programs that support the operation of a computer system
>
> —Leland L. Beck, *System Software: An Introduction to Systems Programming*, 3rd ed. Reading, MA: Addison Wesley, 1997. page 1

Beck's definition is clear and encompasses all programs considered as systems software, such as compilers, loaders, assemblers, text editors, operating systems, and many others. We can classify systems software in two groups: programs to create a software development

environment and programs to create a runtime environment. Examples of programs in each group are mentioned below.

1. Development environment: Some programs that assist programmers in writing application programs are:

   Text editor

   Compilers

   Assemblers

   Linkers

2. Runtime environment: Some programs that create a runtime environment are:

   Operating systems

   Loaders

   Dynamic linkers

   Program libraries

Assemblers and loaders could be considered as the first two pieces of systems software developed to help human beings write programs in a symbolic language and load the **executable program** in the computer memory for execution. **Assembly language** has been in use since the late 1940s; for example, the **Electronic Delay Storage Automatic Calculator (EDSAC)** was a computer built in 1949 at Cambridge University by a team led by Maurice Wilkes. This computer had an assembly language whose operations (opcodes) were encoded using a single letter, a significant step compared to writing programs directly in binary. The assembler and loader were combined into an "assemble-and-go" program that translated the program written in assembly language into executable code and immediately loaded the executable version of the program in memory for execution. The **Fortran** programming language can be considered the first high-level notation for writing programs. Fortran (short for "formula translation") was designed in the late 1950s by John Backus and his team at IBM, and the first compiler was delivered in 1957. With the invention of computer monitors, the necessity of a text editor (full **screen editors**) was a natural challenge, and by the late 1960s, the Control Data Corporation (CDC) had developed a text editor for the operator console. Text editors and compilers made the programming process easier. From the standpoint of creating a runtime environment with a friendly interface, operating systems are the ones that characterize this functionality the most. General Motors implemented the first operating system on an IBM 701 in 1955.

Software Tools for Program Development

In the program developing process, the programmer should focus on solving a problem and expressing the solution of the problem in a programming language. If we provide programmers with the appropriate tools to ease their work, they will be more productive and less distracted

by solving secondary problems. In this section we will briefly present some systems software programs to support programmers in the program development process.

We will assume that programmers understand the syntax of the programming language that they are using to write programs. Then, to write a program, the first tool we have to provide programmers is a text editor. With a text editor, programmers can create and edit their programs at ease. Once they are able to write programs using the text editor, the next systems software program they need to get acquainted with is a compiler. They need to know how to invoke the compiler to translate the program into assembly language or **object code** (binary). If the program is translated into assembly language or if the program was written in assembly language, then programmers need to know what step to follow next to translate the assembly version of their program into object code. This means they need to know how to invoke the assembler. Now you have an idea of how the source program written in a high-level language is translated into an executable version (object code) by using different systems software programs.

In the early days, when they used a "translate-and-go" approach, object code could be loaded into memory and executed. Nowadays, we are able to combine several object codes together into a single module and then load this combined object code for execution. This step, called **linking**, is carried out by the linker or linkage editor. The object code is an executable file that has a header with the program ID and information about the different sections within the file. These sections include the **text section** (code), the **data section**, the **relocation section**, and the **symbol table**, to name a few. The executable file is known nowadays as the **Executable and Linkable Format (ELF)**. When the ELF is loaded into memory for execution, a **process** is created by the operating system and the program is allowed to run. Figure 1.1 illustrates different systems software programs and the way they are related.

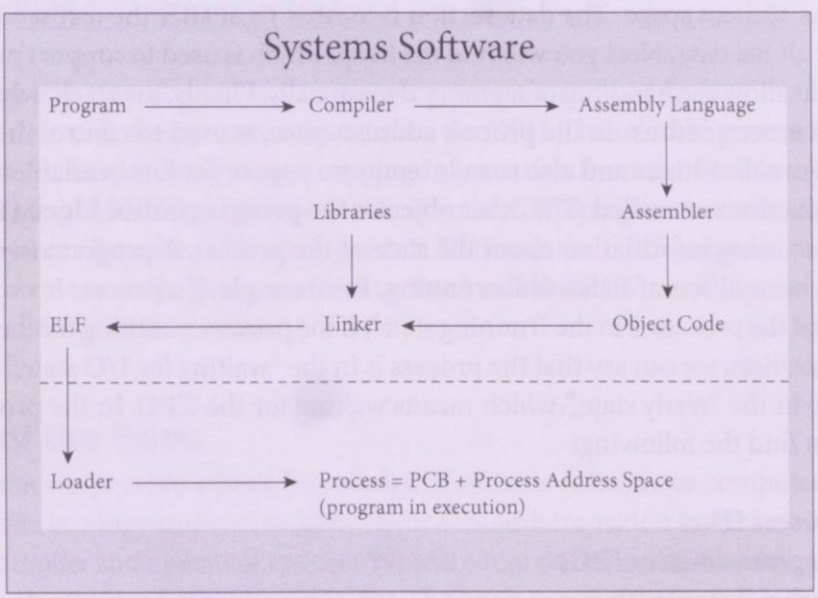


FIGURE 1.1 Systems software road map.

Software to Create Runtime Environments

When thinking about **runtime environments**, the program or set of programs that characterize a runtime environment is the operating system. Operating systems are event-driven programs that create a friendly interface between the user and the computer system. They also handle all system resources efficiently. The **command interpreter** is the layer of the operating system that allows the user to interact with the computer system. A user can type in a command using a command-line interpreter (CLI) or a graphical user interface (GUI), and the operating system will take the appropriate actions. For instance, to invoke a compiler to compile a program, a command must be given to the operating system, and the operating system will load the compiler and allow the compiler to read the user program as a text file. The compiler will translate the program into assembly language, then the operating system might invoke the assembler and allow the assembler to read in the assembly language program, and the assembler will translate the program into object code. To execute the program, we can interact with the operating system and type in a command to tell the operating system to load and execute the program.

To carry out the execution of a program, the operating system has to create two objects in memory per program. One of them is the **process address space**, which is a memory segment divided into four sections:

- **Text** section
- **Data** section
- **Heap** section
- **Stack** section

The text section is used to store the executable program starting at memory address zero in the process address space. The data section is located right after the text section, and it is used to store global data. Next you will find the heap, which is used to support programming languages that allow the handling of memory dynamically. Finally, the stack, which begins at the highest memory address in the process address space, is used to control the calling and returning from subroutines and also to save temporary space for local variables and parameters when functions are called. The other object is the **process control block (PCB)**, a data structure containing information about the state of the process. A program in execution (a process) can be in different states while running. For example, if a process is using the CPU, we can say that the process is in the "running state"; if the process is waiting for the completion of an I/O operation, we can say that the process is in the "waiting for I/O state." The process could be also in the "ready state," which means waiting for the CPU. In the process control block, we can find the following:

- The **process ID**
- The **program counter (PC)**
- The **stack pointer (SP)**

- The **state** of the process (for instance, running or ready)
- Contents of the **CPU registers** at the moment the process left the CPU
- And some other relevant information that allows the operating system to keep track of each process. All the information stored in the PCB is called the context.

Figure 1.2 illustrates the relationship among the object code, the process address space, and the process control block. Observing the figure in detail, you will find that the text and data sections of the object code (executable file) are copied from the object code into the process address space.

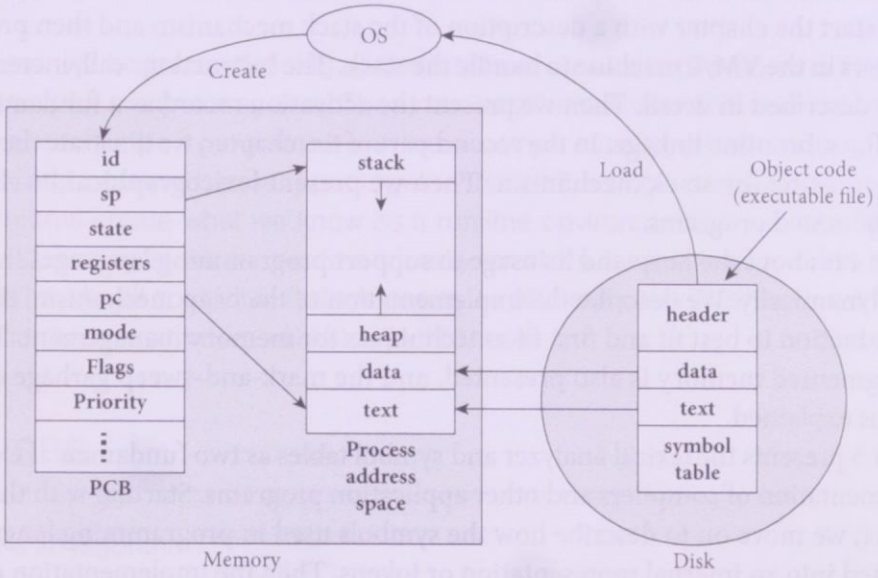


FIGURE 1.2 Process creation.

When there are several processes in memory, they are competing for CPU time, and the operating system will choose one of them to use the CPU. The selected process is dispatched (CPU assigned). **Dispatching** means copying information from the PCB into the CPU; for instance, copying the PCB program counter into the CPU program counter or copying the PCB stack pointer into the CPU stack pointer. This way the CPU program counter will be pointing in memory to the program (text section) and the CPU stack pointer will pointing to the stack of the selected process.

Outline of the Book

This book aims to provide a basic knowledge of systems software to computer science and information technology students to allow them a smooth transition toward more specialized courses in compiler and operating systems. We will equip students with a background sufficient to delve into those areas with little difficulty. The book is organized as follows.

Chapter 2 presents the processor as an instruction interpreter of assembly language. Computer organization concepts are given, and the repertoire of instructions, or instruction set architecture (ISA), of a virtual machine is explored in detail. We start out the chapter with a review of positional number systems and data representation. Then we present the architecture of a register virtual machine, which we call VM/0, to show students the relationship between assembly language and computer organization, which is fundamental to writing assemblers and compilers. Toward the end of the chapter, we give a smooth introduction to the process address space.

Chapter 3 deals with the stack concept as a mechanism to implement the invocation of subroutines or functions and to control the return from the subroutine to the caller environment. We start the chapter with a description of the stack mechanism and then present two new registers in the VM/0 machine to handle the stack. The instructions call, increment, and return are described in detail. Then we present the activation record as a fundamental data structure for subroutine linkage. In the second part of the chapter, we illustrate the handling of recursion using the stack mechanism. Then we present lexicographical levels and the handling of nested programs.

Chapter 4 is about the heap and its usage to support programming languages that handle memory dynamically. We describe the implementation of the heap mechanism, then give a brief introduction to best fit and first fit as techniques for memory management. The problem of fragmented memory is also presented, and the mark-and-sweep garbage collection approach is explained.

Chapter 5 presents the lexical analyzer and symbol tables as two fundamental elements in the implementation of compilers and other application programs. Starting with the concept of alphabet, we move on to describe how the symbols used in programming languages can be converted into an internal representation or tokens. Then the implementation of symbol tables is explained in detail.

Chapter 6 gives you a basic knowledge of parsing and code generation. These two concepts are central to the development of compilers, and understanding them will allow you to delve into the compiler arena in a comfortable way. Examples of parsing and generating code for different programming statements are given.

Chapter 7 is about assemblers and loaders. Assemblers allow you to understand how assembly language is translated into an executable file, and by knowing about loaders, you will be able to understand the way the operating system loads programs for execution. Linkers as programs that create an executable linkable file from several object files are briefly presented.

Chapter 8 presents the interrupt mechanism as a fundamental concept to help readers understand the way user programs communicate with the operating system to request service. The interrupt mechanism also provides a protection mechanism to assist the operating system in detecting and stopping malicious or careless user programs from invading other user address spaces or even the operating system itself. The interrupt mechanism also provides an interface between user programs and the operating system, which allows user programs to request service from the operating system.

Chapter 9 presents the concept of process as a program in execution. Process and thread implementation issues are described. We also present the support required to implement a multithreading environment. A brief introduction to handling multiples treads on a multicore is also given.

Chapter 10 deals with implementation issues of concurrent execution of sequential programs and process synchronization. Hardware and software mechanisms are studied in detail. We conclude the chapter with an introduction to the message passing mechanism.

SUMMARY

In this chapter we began by defining systems software as a set of programs that support the operation of a computer system. Then we classified systems software in two categories: programs to support programmers in the development of applications programs, and another set of programs, basically put together under the single name operating system. These programs create what we know as a runtime environment to execute application programs. Operating systems create a friendly interface for the user and the computer system and handle all computer system resources efficiently. We illustrated, using a block diagram, the road map that we will follow in this book.

EXERCISES

1. What is an algorithm?
2. What is a program?
3. What is a compiler?
4. What is a text editor?
5. What is an assembler?
6. What is assembly language?
7. Does object code = assembly language?
8. When the operating system dispatches a process, name two fields of the PCB that are copied in the CPU.
9. What is an operating system?
10. What is a loader?
11. Are assemblers and compilers translators?
12. We can refer to all the information stored in the PCB by a single name. What is that name?

13. Fortran is considered the first implementation of a high-level language. Is it still in use?
14. Name five programs that are considered to be system software.
15. What is a system programmer?

Bibliographical Notes

An interesting work on the benefits of programming methodology is described by Donald Knuth in.[1] Programs considered to be systems software have been around since the late 1940s as tools to ease the programming process.[2,3] David Barron presents a good monograph describing assemblers and loaders.[4] An in-depth source for implementing assemblers and loaders is presented by David Salomon.[5] Leland L. Beck gives a very good and in-depth description of various types of systems software in his system software book.[6] The book could be considered the first on systems software as we know it nowadays. The Fortran compiler was a major step in the development of systems software to ease the programming process by allowing programmers to write programs in a high-level language.[7] Another valuable tool to help programmers write programs was the full-screen editor developed by the Control Data Corporation in the late 1960s for the operator console. The General Motors–North American Aviation input/output system (GM-NAA I/O), also known as GM OS, was the first operating system developed, and it was implemented on an IBM 701.[8] It was a batch system, which allowed a program to run as soon as the one being executed had run to completion.

Bibliography

1. D. E. Knuth, *Literate Programming*, The Computer Journal, Vol. 27, No. 2, 1984.
2. M. V. Wilkes and W. Renwick, "The EDSAC, an Electronic Calculating Machine," *Journal of Science and Instrumentation* 26 (1949), p. 385.
3. M. M. Wilkes, "The EDSAC Computer," in *Joint AIEE-IRE Computer Conference Review of Electronic Digital Computers*, 1951.
4. D. W. Barron, *Assemblers and Loaders*, 3rd ed. Elsevier North-Holland, 1978.
5. D. Salomon, *Assemblers and Loaders*. Ellis Horwood, 1993.
6. L. L. Beck, *System Software: An Introduction to Systems Programming*, 3rd ed. Reading, MA: Addison Wesley, 1996.
7. J. W. Backus et al., "The FORTRAN Automatic Coding System," in *Western Joint Computer Conference*. Los Angeles, 1957.
8. R. L. Patrick, *General Motors/North American Monitor for the IBM 704 Computer*. Chicago, IL: RAND Corporation, 1987.

Chapter 2

The Processor as an Instruction Interpreter

The world was so recent that many things lacked names, and in order to indicate them it was necessary to point.

—Gabriel Garcia Márquez

One Hundred Years of Solitude

INTRODUCTION

The role of a processor (CPU) in any computer system is to execute programs stored in memory. These programs are represented as a sequence of zeros and ones. The processor must discern program instructions from data values. Human beings have developed programming notations, known as programming languages, to write programs to instruct computers to follow a series of commands to solve problems using a computer. One of these programming languages is assembly language, also known as symbolic language.

The processor is basically an interpreter of assembly language instructions, and in this chapter we will present the organization of a tiny computer and a set of assembly language instructions that our tiny processor accepts. This repertoire of instructions is called the instruction set architecture (ISA). For each instruction, we will discuss its instruction format and how it is encoded in binary. Finally, we present the concept of process address space to visualize the way the operating system assigns and organizes memory space for program execution.

CHAPTER OBJECTIVES

- To review positional number systems used by programmers and computers to represent data and instruction.
- To discuss the necessity of knowing computer organization to understand assembly language.
- To explore how assembly language works.
- To describe the concept of process address space.

VOCABULARY

This is a list of keywords that herald the concepts we will study in this chapter. They are organized in chronological order and will appear in the text in bold. We invite you to take a look at these keywords to find out which ones you are familiar with.

Decimal number
Hexadecimal number
Binary number
P-code
Algorithm
Program
Programming language
Memory
Input/output devices
Processing element (PE)
Central processing unit (CPU)
Instruction set architecture (ISA)
Assembly language
Register
Register file (RF)
Instruction register (IR)
Control unit (CU)
Arithmetic logic unit (ALU)
Condition code (CC)
Flags
Bit
Byte
Half byte
Word
Random access memory (RAM)
Memory address register (MAR)
Memory data register (MDR)
Program counter (PC)
Instruction cycle
Fetch
Decode
Execute
Identifier
Memory location
Variable
Assembler
System call
Operating system (OS)
Standard input (stdin)
Standard output (stdout)
Assembler directives
Text
Data
End
Instruction format
Global data pointer (GDP)
Object code
Process
Program in execution
Process address space
Stack segment
Heap segment
Data segment
Text segment

ACTIVATING PRIOR KNOWLEDGE

In this section we will present a series of activities. In some of them you can choose one or more options. Sometimes, if you do not agree with the given answers to choose from, you will be allowed to give your own answer. By the way, this is not a test.

1. The number 10 is:

   A decimal number ☐
   A binary number ☐
   A hexadecimal number ☐
   All are valid answers ☐

2. What is the language that computers execute?

   Python ☐ Go ☐ Java ☐ Assembler ☐ Binary ☐

3. Assume that ADD can be encoded as 0001 and we can use three registers identified as A, B, and C. If A is encoded as 0011, B as 0010, and C as 0001, write the instruction ADD C B A using zeros and ones.

4. What is the highest value that can be expressed with 8 bits?

   8 ☐
   64 ☐
   16 ☐
   If the answer is not given above, please give yours. _____

5. A compiler is a program that translates:

   From C++ to binary ☐
   From C++ to machine language ☐
   From C++ to assembly language ☐
   From C++ to object code ☐

6. To execute a program, the OS must load the program from disk. Where does the operating system load the program?

   In the CPU ☐
   In memory ☐
   50 percent in memory and 50 percent in the CPU ☐

7. Skim the chapter and pay attention to the words in **bold**. Count how many of those words you are familiar with.

On Naming and Data Representation

Naming is a gift that allows human beings to identify real and abstract objects. Furthermore, naming permits us to synthesize a concept or a definition of a complex concept in a single word. Without a naming scheme only ostensive references would be possible, which would make the handling of information in any knowledge domain extremely difficult. Therefore, we need to use a naming scheme to identify the information units and mechanisms to store and handle objects in a computer system.

Probably, the first abstract objects or symbols we assign names to are the **decimal numbers**. The decimal number system is a positional system that uses ten symbols to represent numbers. They are **0, 1, 2, 3, 4, 5, 6, 7, 8, and 9**. Each one of these digits has a name that we use for human communication. For example, the symbol **3** is called **three**. These basic decimal digits can be grouped into a string of digits to construct other numbers. As the decimal system is a positional number system, the same digit has a different value depending on its position in the string. For instance, in the number 777 the seven on the right has a value of 7, the one in the middle has a value of 70, and the one in the left has a value of 700.

TABLE 2.1 REPRESENTING A NUMBER IN THE DECIMAL SYSTEM

Position in the string	5	4	3	2	1	0
Position value in power of 10	10^5	10^4	10^3	10^2	10^1	10^0
Position value	100,000	10,000	1,000	100	10	1
Decimal number	0	0	0	7	7	7

From Table 2.1 we can deduce the value of a decimal number. It can be obtained by multiplying each symbol in the string by its position value and then adding up all the multiplication carried out, as shown below:

$$777_{10} = (7 \times 100) + (7 \times 10) + (7 \times 1) = 777_{10}$$

This can be rewritten using the power notation using 10 as the base of the system (ten symbols), where the exponent indicates the digit position:

$$777_{10} = (7 \times 10^2) + (7 \times 10^1) + (7 \times 10^0) = 777_{10}$$

Similarly, we could define another set of 16 symbols and create the **hexadecimal number** system. This system has 16 symbols. For the first ten symbols we will borrow the ten symbols of the decimal system. For the others six symbols needed, any symbol could be selected, but the first six letters of the alphabet will do the job. Let us say then our symbols will be **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F**. Since we are used to working with numbers, numerical values are given to each letter symbol A, B, C, D, E, and F, as presented in Table 2.2.

TABLE 2.2 REPRESENTING SYMBOLS AND VALUES IN THE HEXADECIMAL SYSTEM

Symbol	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

As the hexadecimal system is a positional system as well, each symbol will have a different value depending on its position in the string. In this case the base of the system is 16; therefore, powers of 16 will be used. Applying the same rules we used for decimal symbols, we can calculate, using Table 2.3, the decimal value of the hexadecimal number 1AF.

$$\begin{aligned} 1AF_{16} &= (1 \times 16^2) + (10 \times 16^1) + (15 \times 16^0) \\ &= (1 \times 256) + (A \times 16) + (F \times 1) = 431_{10} \end{aligned}$$

TABLE 2.3 REPRESENTING A NUMBER IN THE HEXADECIMAL SYSTEM

Position in the string	5	4	3	2	1	0
Position value in power of 10	16^5	16^4	16^3	16^2	16^1	16^0
Position value	1,048,576	65,536	4,096	256	16	1
Hexadecimal number	0	0	0	1	A	F

You can use the prefix **0x** to indicate that the number is in hexadecimal notation. Therefore, you can write down 0x1AF instead of $1AF_{16}$ in the above equation, and it has the same meaning.

$$\begin{aligned} 0x1AF &= (1 \times 16^2) + (10 \times 16^1) + (15 \times 16^0) \\ &= (1 \times 256) + (A \times 16) + (F \times 1) = 431_{10} \end{aligned}$$

We could define another positional system involving only two symbols, say **0** and **1**, and create a **binary number** system. Binary numbers are central to computer systems because that is the language computers use. The first evidence of the usage of binary numbers is found in China circa 2500 BCE, in the ***I Ching***, or ***Book of Changes***. The *I Ching* uses an unbroken line to represent "heaven" and a broken line to represent "void." The two symbols are grouped into trigrams. A trigram is a set of three lines in which any one of the three lines could be broken or unbroken. An example of a trigram is ☰. In 1703 the German philosopher and mathematician Gottfried Wilhelm von Leibniz (1646–1716) presented his work to the European scientific community to show the correspondence between the unbroken line and the digit 1 and the broken line and the digit 0. In his work he also showed formally that trigrams represented a positional binary number system and illustrated the development of binary arithmetic operation. However, there is evidence that the British mathematician Thomas Heriot (1560–1621) had experimented with binary arithmetic before Leibniz. By the way, Heriot invented the symbols for the relational operators "greater than" (>) and "less than" (<).

The binary system only supports two symbols, making its base 2. Following a similar argument as the one used for the hexadecimal system, we can figure out that the decimal number 5 is represented in binary as **101**. You can verify this by using Table 2.4 and observing that each binary number is multiplied by its positional value:

$$\begin{aligned} 101_2 &= (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= (1 \times 4) + (0 \times 2) + (1 \times 1) = 5_{10} \end{aligned}$$

The three positional systems described above are central to information technology and computer science because decimal numbers are used by human beings, binary is used by computers, and hexadecimals can be thought of as a short notation to deal with strings

TABLE 2.4 REPRESENTING A NUMBER IN THE BINARY SYSTEM

Position in the string	5	4	3	2	1	0
Position value in power of 10	2^5	2^4	2^3	2^2	2^1	2^0
Position value	32	16	8	4	2	1
Binary number	0	0	0	1	0	1

of binary numbers. Let us illustrate this by expressing the number 431 using 16 binary digits and 4 hexadecimals digits.

$$431_{10} = 0000\ 0001\ 1010\ 1111_2 = 0x01AF$$

TABLE 2.5 REPRESENTING 16 DISTINCT MEMORY ADDRESSES IN BINARY (HALF BYTE), HEXADECIMAL, AND DECIMAL

Binary	hexadecimal	decimal
0000	0x0	0
0001	0x1	1
0010	0x2	2
0011	0x3	3
0100	0x4	4
0101	0x5	5
0110	0x6	6
0111	0x7	7
1000	0x8	8
1001	0x9	9
1010	0xA	10
1011	0xB	11
1100	0xC	12
1101	0xD	13
1110	0xE	14
1111	0xF	15

If you observe carefully, you will realize that each hexadecimal digit can be represented in binary using 4 binary digits. Table 2.5 can be used to translate from binary to hexadecimal and vice versa any time we are dealing with memory locations or need to observe the contents of a specific memory location. It is recommended to work with hexadecimal digits instead of binary digits.

0000	0001	1010	1111
↓	↓	↓	↓
0	1	A	F

A P-code Register Machine

P-code, or pseudo code, is the name given to the instruction set of a virtual machine or software interpreter. In this chapter we will present VM/0, a p-code architecture based on the interpreter proposed by Niklaus Wirth in the implementation of PL/0. The main difference between VM/0 and Wirth's p-code is that in VM/0, a register file is used to carry out arithmetic and relational operations. A subset of the VM/0 assembly language will be presented in this chapter and will be extended in chapter 3.

The words **algorithm**, **program**, and **programming language** are used quite often when we try to solve a problem using a computer. We will define them to set up a common understanding of the meaning of these words. An **algorithm** is a set of steps to be performed to solve a specific problem. If you intend to solve a problem using a computer, you will need to instruct the computer, and to do this, you have to write a program. Therefore, we can define a **program** as an algorithm expressed in a programming language. A **programming language** is a notation (or "artificial language") created to instruct the computer and to allow programmers to understand each other's program.

Computer Organization

Since John von Neumann conceived the idea of a stored-program computer in 1945, a computer consists basically of three units: **memory**, **input/output devices**, and a **processing element (PE)**, also known as a **central processing unit (CPU)**. Nowadays, there are other types of processing elements designed for specific purposes, such as **graphic processing unit (GPU)**, **tensor processing unit (TPU), and quantum processing unit (QPU)**. When designing a CPU, we are creating an instruction interpreter capable of executing a set of instructions or commands generally known as the **instruction set architecture (ISA)**. You can look at the ISA from two different perspectives: from the hardware standpoint, the CPU uses zeros and ones; and from the programmer standpoint, without an ISA, programmers would be doomed to program the computer in binary. **Assembly language**, or symbolic language, was created, as any other programming language, to allow programmers to understand each other's program and to instruct the computer to executed programs. To give you an idea of programming in assembly language, let us start describing VM/0 CPU. There are several **registers** in VM/0 CPU. Each register is a small, high-speed logic circuit capable of storing a binary number, and we can think of it as a container. Some of these registers are grouped together under the name **register file (RF)**, and these are the sole registers programmers can use, through assembly language instruction, to manipulate data. For a register file of four registers, we can denote them as R[0], R[1], R[2], and R[3], where R stands for a register and the number in square brackets identifies a register in the register file. For example, let us assume that we need to add two numbers, say 17 and 23, and the two numbers are stored in register R[0] and R[3] respectively, and the resulting value of the arithmetic operation will be stored in R[2]. We can carry out the operation R[2] = R[0] + R[3] using the assembly language instruction ADD this way:

```
ADD R[2], R[0], R[3]    ; programmer level
```

Note: The ";" is used to indicate the beginning of a comment.

This instruction should be read as follows: take the contents of register zero (whose value is 17) and register three (whose value is 23), add them up, and store the resulting value (40) in register two. It is worth noting, that R[0] and R[3] are the source registers, and R[2] is the

target or destination register. For encoding R[2] = R[0] + R[3] in binary using 16 binary digits, we can expect something like:

```
0000 1000 1100 0001     ; computer level
0x08C1                  ; in hexadecimal
```

Similarly, the numbers stored in R[0] and R[3] could be used to compute R[2] = R[0] − R[3] using the assembly language instruction SUB as follows:

```
SUB R[2], R[0], R[3]    ; programmer level
```

Another register in the CPU is the **instruction register (IR)**. Programs written in assembly language do not allow programmers to manipulate IR—only the CPU hardware can do that. When the IR receives an instruction, such as ADD R[2], R[0], R[3] from memory, it forwards the opcode, ADD, to the **control unit (CU)**, and the CU decodes the opcode (identifies the instruction) and initiates the necessary steps to execute that instruction. The CPU control unit is hardwired to decode and execute all instructions of the ISA. The operation R[2] = R[0] + R[3] is carried out in the **arithmetic logic unit (ALU)**; this unit is commanded by the control unit to execute all types of arithmetic, relational, and logic operations the computer supports. Figure 2.1 depicts VM/0 executing the operation ADD R[2], R[0], R[3].

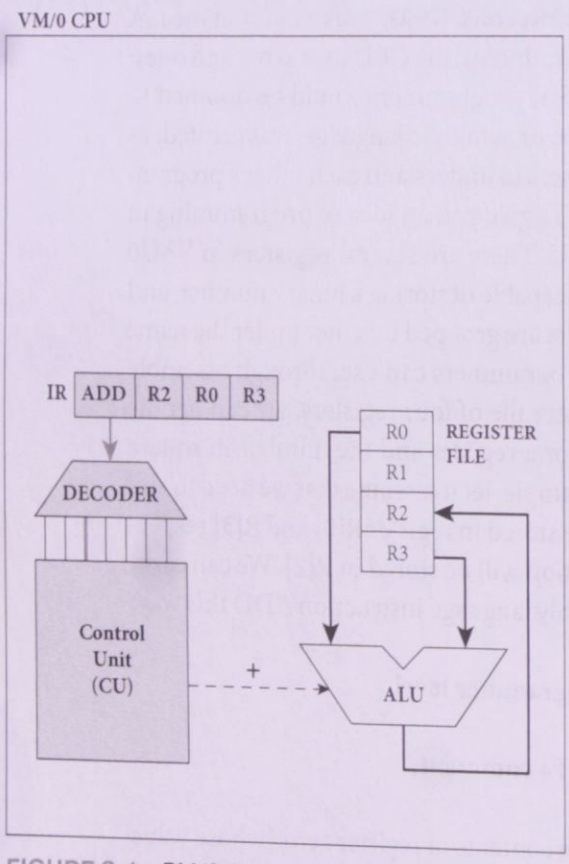


FIGURE 2.1 PM/0 CPU executing instruction ADD.

We just gave an example of an arithmetic operation, but the CU is able to execute other types of operations, such as relational operations; for instance, comparing two numbers to find out if they are *equal*. To explain the way this can be carried out, we need to introduce the instruction **EQ** (test for equality) and show how it works using **condition codes (CCs)**. Condition codes are associated with the ALU and they can be tested after execution of an arithmetic, relational, or logical operation, for knowing their status, which can change as a side effect of the execution of the instruction. There are three **flags** in the CC register in VM/0-CPU (a flag can be thought of as a single-bit register): N (negative), Z (equal to zero), and P (positive). They are extremely useful for changing the control flow of a

program. A condition code is set to 1 if a condition is met; otherwise, its value is zero. Assuming R[0] = 21 and R[3] = 21, we can figure out that condition code Z is set to 1, once the compare register instruction (EQ) has been executed because the condition R[0] = R[3] holds. Figure 2.2 illustrate the execution of the instruction EQ.

```
EQ R[0], R[3]    ; programmer's level
```

The way relational operations work is by subtracting the contents of the two registers involved in the instruction, and depending on the resulting value, condition codes are set out. In the former example, as both registers have the same value, then R[0] – R[3] = 0, and therefore condition code Z is set to 1. These relational instructions affect condition code flags. Table 2.6 shows the way CCs are set depending on the relational or arithmetic operation executed.

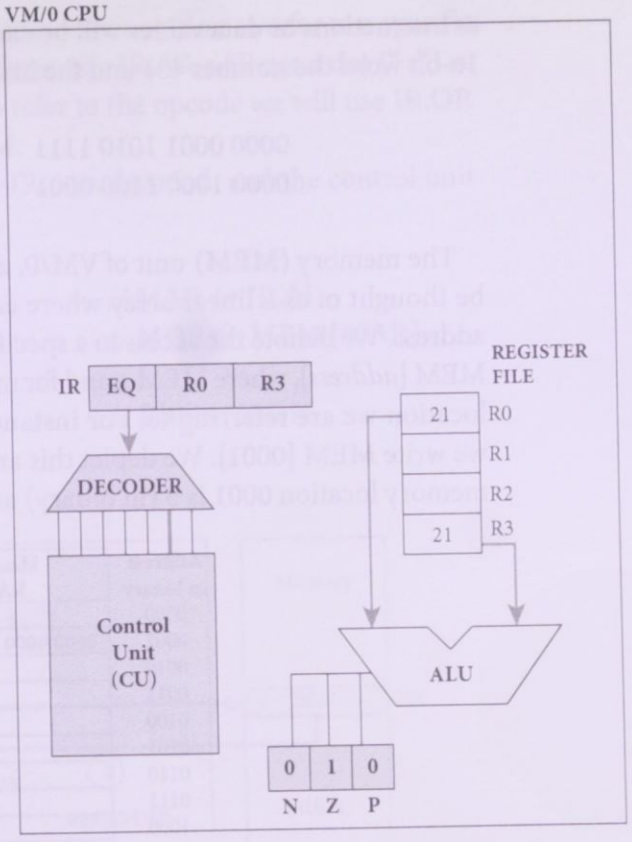


FIGURE 2.2 VM/0 CC executing instruction EQ.

Memory Organization

The storing and handling of information in memory can be carried out at different levels. In VM/0, as in any other computer system, the basic information unit is the **b**inary dig**it**, widely known as a **bit**, which allows us to represent the two symbols of the binary number system (0 and 1). Bits can be grouped together to form other information units that convey more information, like **bytes**, **half bytes**, and **words**. A byte is a sequence of 8 bits capable of storing $2^8 = 256$ different values, and a half byte is represented with a sequence of 4 bits. A word could have variable length, say 16, 32, or 64 bits, depending on technology, and it gives us information about the size of each memory location. In VM/0 a word consists of a sequence of 16 bits. Each memory location contains either an instruction or a data value, and the interpretation of those 16 bits (the word)

TABLE 2.6 CONDITION CODES SETTINGS

Description	Condition codes
Test for equality	Set Z = 1
Not equal	Set Z = 0
Less than	Set N = 1
Less than or equal to	Set Z =1 or N = 1
Greater than	Set P = 1
Greater than or equal to	Set Z =1 or P = 1

as instructions or data values will be carried out by the CPU or processor. For example, in a 16-bit word the number 431 and the instruction ADD R[2], R[0], R[3] can be represented as:

```
0000 0001 1010 1111     ; this is the data value 431 in binary
0000 1000 1100 0001     ; this is instruction ADD in binary
```

The memory (**MEM**) unit of VM/0, also known as **random access memory (RAM)**, can be thought of as a linear array where each memory location or word has a label called its address. We denote the access to a specific memory location by using the following notation MEM [*address*], where MEM stand for memory and *address*, a number, indicates the memory location we are referring to. For instance, to access the content of memory location 0001 we write MEM [0001]. We depict this arrangement in Figure 2.3. Notice that the content of memory location 0001 is 5 (in binary) and the content of MEM [1101] is 431 (in binary).

Address in binary	Memory RAM	Address hexadecimal	Address decimal
0000		0x0	0
0001	**0000 0000 0000 0101**	**0x1**	**1**
0010		0x2	2
0011		0x3	3
0100		0x4	4
0101		0x5	5
0110		0x6	6
0111		0x7	7
1000		0x8	8
1001		0x9	9
1010		0xA	10
1011		0xB	11
1100		0xC	12
1101	**0000 0001 1010 1111**	**0xD**	**13**
1110		0xE	14
1111		0xF	15

FIGURE 2.3 Representing 16 distinct memory addresses in binary (half byte), hexadecimal, and decimal.

There are two CPU registers used to access memory; they are known as **memory address register (MAR)** and **memory data register (MDR)**. For selecting a memory location, MAR is used. MDR has a bidirectional connection with main memory, and this allows it to receive in a CPU register the contents of a specific memory location selected by MAR or to store a value from a register into a memory location pointed to by MAR. These two registers allow us, using assembly language instruction, to move data and instructions back and forth between the CPU and memory. This occurs at the CPU level, and programmers do not have to bother about these details. Figure 2.4 shows a detailed view of the execution of the instruction **LOD R[0], 0, M**. In this instruction, LOD means load or copy data from memory into a CPU register; the first operand indicates the register receiving the data; the second value, a zero, means unused (n/a), but it is mandatory to use it (you will find out why in chapter 3), and **M** stands for a memory address (identifies a memory location). Before explaining how the

instruction is executed, we must clarify some details about the notation to refer to a specific field in a VM/0 instruction once the instruction resides in the IR. We will use a dot (".") as a selector. For example, the IR has four fields, and to refer to the opcode we will use IR.OP. Likewise, to refer to the address M we will write IR.M.

Once the instruction **LOD R[0], 0, M** reaches the IR, it is decoded, and the control unit carries out three steps to execute the instruction:

1. Memory address **M** is copied in **MAR**. // **MAR ← IR.M**
2. **MAR** selects memory location **M** and sends its content to **MDR**. // **MDR ← MEM [MAR]**
3. The content of **MDR** is copied in **R[0]**. // **R[0] ← MDR**
4. Back to Fetch

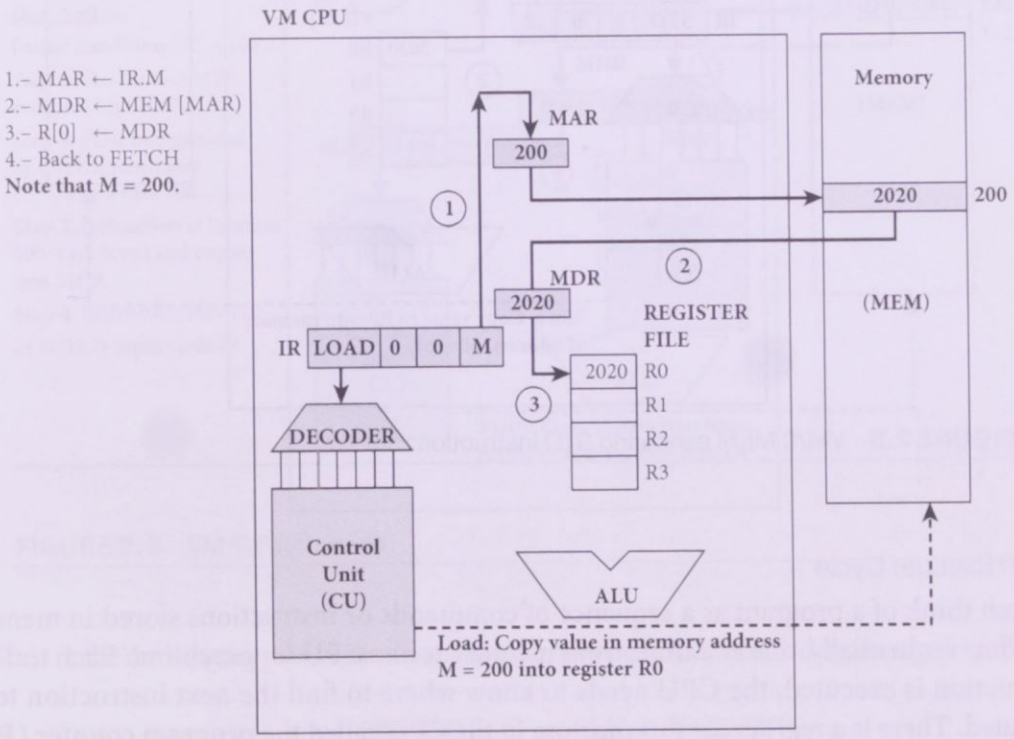


FIGURE 2.4 VM/0 executing LOD instruction.

Sometimes we need to store the content of a register, say R[0], in a memory location, in which case we must use the instruction store (**STO**). When instruction **STO R[0], 0, M** reaches the IR, it is decoded, and the control unit will carry out the following steps:

1. The content of memory address **M** is copied in **MAR**. // **MAR ← IR.M**
2. The content of R[0] is copied in **MDR**. // **MDR ← R[0]**
3. **The content of MDR** is copied into memory address **M**. // **MEM[MAR] ←MDR**
4. Back to Fetch

Figure 2.5 shows the execution of the store instruction. Observe in the figure t
encircled numbers match steps 1, 2, and 3 of the execution of the instruction STO as de
above.

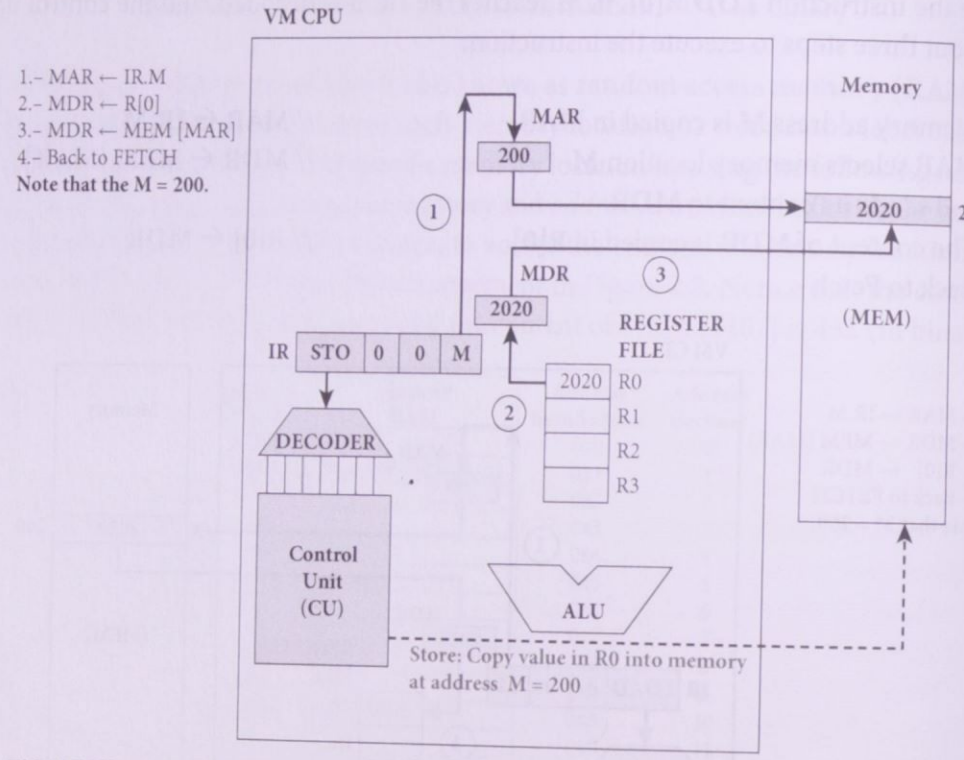


FIGURE 2.5 VM/0 MEM executing STO instruction.

Fetch Execute Cycle

We can think of a program as a sequence of commands or instructions stored in me
that flow sequentially, one at a time, from memory to the CPU for execution. Each tir
instruction is executed, the CPU needs to know where to find the next instruction
executed. There is a register for this purpose in the CPU called the **program counter (**
which points to the next instruction to be fetched for execution. The reason we call it
because instructions in memory are stored sequentially in contiguous memory locat
therefore, each time an instruction is retrieved from memory and sent to the CPU for e
tion, the PC is incremented by "one" to point to the next instruction in the sequence, w
is stored in the next memory location. Thus, the PC behaves just like a counter. This
general how the CPU controls the flow of instructions from memory to the CPU. The
steps required to execute an instruction or **instruction cycle** are **fetch, decode,** and **exe**
In **fetch**, an instruction is copied from memory into the CPU instruction register, an
PC is incremented by one to point to the memory location where the next instruction

executed resides. Figure 2.6 illustrates the fetch cycle at the machine level. These are the steps carried out in Fetch:

```
FETCH:   MAR ← PC
         PC  ← PC + 1
         MDR ← MEM[MAR]
         IR  ← MDR
```

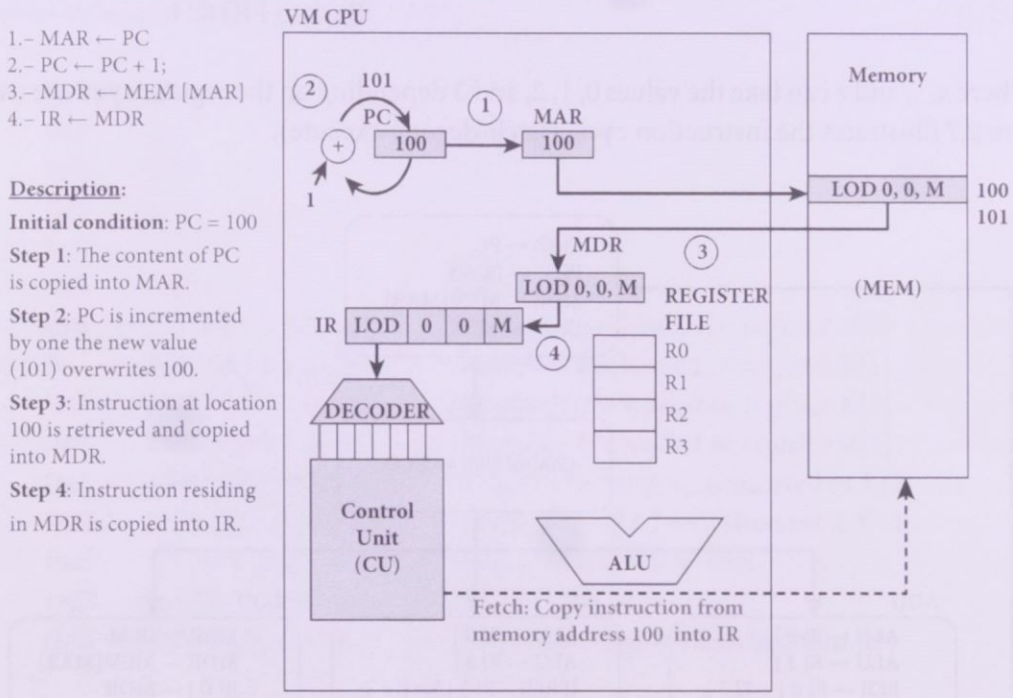


FIGURE 2.6 VM/0 Fetch cycle.

Then in the **decode** step, the control unit uses the opcode (IR.OP) and identifies the instruction.

```
DECODE: Control Unit ← IR.OP
```

Once the instruction has been identified, the CU immediately initiates all required actions to execute the identified instruction. For example, for the instructions ADD, EQ, and LOD, we have:

```
If IR.OP = ADD    Then   ALU ← R[y]
                         ALU ← R[z]
                         R[x] ← R[y] + R[z]
                         Back to FETCH
```

```
If IR.OP = EQ        Then   ALU ← R[x]
                            ALU ← R[y]
                            If R[x] = R[y] then set Z ← 1
                            Back to FETCH

If IR.OP = LOD       Then   MAR ← IR.M
                            MDR ← MEM [MAR]
                            R[x] ← MDR
                            Back to FETCH
```

Where x, y, and z can take the values 0, 1, 2, and 3 depending on the registers you are using. Figure 2.7 illustrates the instruction cycle (fetch-decode-execute).

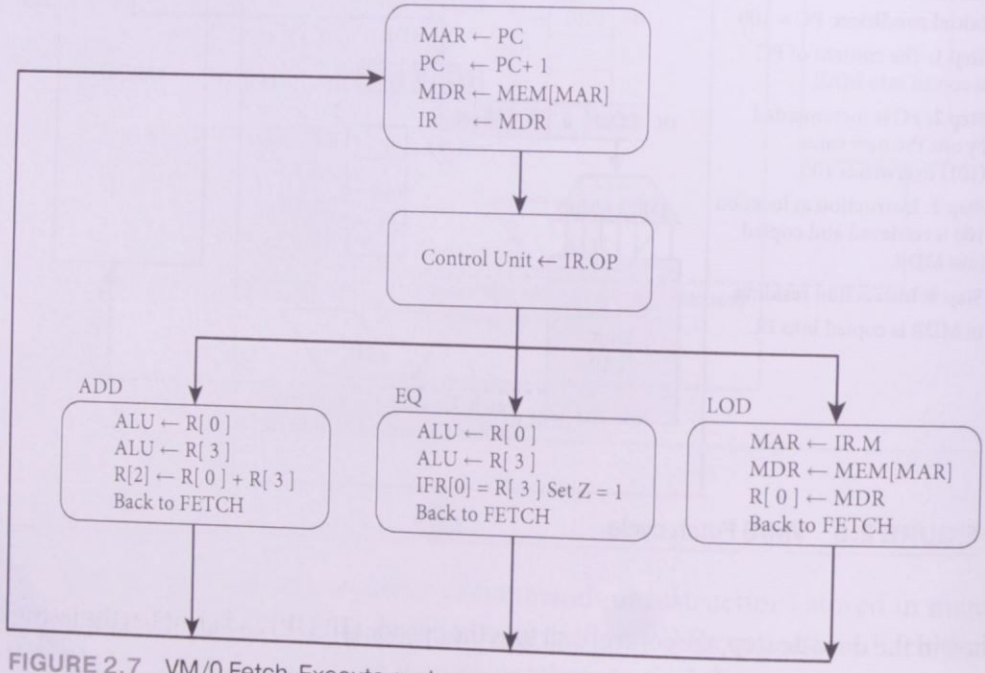


FIGURE 2.7 VM/0 Fetch-Execute cycle.

As you can observe in the fetch cycle, each time an instruction is retrieved from memory, the PC is incremented by one to point to the instruction that resides in the next memory location. Nevertheless, there are some instructions that allow the disruption of sequential execution by overwriting the PC before the next instruction cycle begins, and in this way different and more interesting ways to control the flow of instructions toward the CPU can be implemented.

Assembly Language Programming

A good example of an instruction that overrides the program counter thanks to the use of condition codes is **J**um**P** on **C**ondition (JPC). Another instruction that overwrites the PC but without checking out condition code flags is jump unconditional (JMP). We will put together some of the instruction already explained to write a small program in assembly language to illustrate the usage of JPC 0, 0, M, where M is a memory address. This instruction checks on condition code flags, and if any one of them is equal to 1 then the PC is overwritten by copying IR.M into the PC (PC ← IR.M). Note that in the next example, memory addresses are given in hexadecimal

```
0x0   0
0x1   5
0x2   3
0x3   1
0x4
0x5
0x6   LOD R [ 0 ], 0, 0x2      ; Number 3 is loaded in register R[0]
0x7   LOD R [ 1 ], 0, 0x3      ; Number 1 is loaded in register R[1]
0x8   LOD R [ 2 ], 0, 0x0      ; Number 0 is loaded in register R[2]
0x9   LOD R [ 3 ], 0, 0x1      ; Number 5 is loaded in register R[3]
0xA   ADD R[2], R[2], R[3]     ; Successive sums accumulated in R[2]
0xB   SUB R[0], R[0], R[1]     ; If (R[ 0 ] - R[ 1 ] = 0) then set Z ← 1
0xC   JPC 0, 0, 0xE            ; If Z = 1 then PC ← 0xE
0xD   JMP 0, 0, 0xA            ; PC ← 0xA
0xE   STO R[2], 0, 0x4         ; Store resulting value in location 0x4
```

In this program, the number before the instruction indicates the memory location where the instruction resides. The PC points to the first instruction (PC = 0x6), and a single 0 in any instruction field means that the operand is not used by the CPU. This program multiplies two numbers (5 x 3) using successive additions. In instructions 0x6 to 0x9, registers are loaded with the initial values stored in locations 0x0 to 0x3. The initial value of register R[2] is zero. The instruction at address 0xA will add 5 to register R[2] each time it is executed, and the resulting value is stored back in R[2]. The value 3 stored initially in R[0] will be used to control the loop. Each time the SUB instruction is executed, R[0] is decremented by one, and eventually R[0] will reach the value zero and will set condition code Z to one (Z ← 1). Instruction JPC checks whether Z is equal to one, and if it is, address 0xE is copied into the PC, and the next instruction to be executed will be the one at location 0xE. But if the condition is not met (Z ≠ 1), the address 0xA is copied into the PC, and the program remains in the loop.

In the next example you can observe the usage of JPC after using the instruction EQ. As the two registers have the same value, EQ will set Z to one. Then JPC realizes that Z = 1 and makes PC ← 0x9, in this way skipping any instruction stored in location 0x8.

Memory Address	Memory (RAM)	
0x0		
0x1	8	DATA
0x2	8	
0x3		
0x4	LOD R[0], 0, 0x1	TEXT (code)
0x5	LOD R[1], 0, 0x2	
0x6	EQ R[0], R[1], 0	
0x7	JPC 0, 0, 0x9	
0x8	**ADD R[0], R[0], R[1]**	
0x9		

Equipped with this set of assembly language instructions, you can command the CPU to execute them. But what about the handling of data? We will introduce a pseudo-instruction **WORD** that will help you handle data in VM/0 assembly language. We will start out with an example to show you the usefulness of this pseudo-instruction. Imagine you have two values stored in two different memory locations (or cells)—and as you know, inside a computer everything is represented in binary. For example:

memory address	memory contents (variables)
000000000001	0000000000001001
000000000010	0000000000011000

WORD permits the association of names or labels with a memory location. These names are also known as **identifiers.** An identifier allows programmers to refer to a **memory location** using a name instead of strings of 0s and 1s. As the content of memory locations might change as the program runs, memory locations are also known as **variables**. WORD allows programmers to reserve a memory word and define the initial value for that specific memory location. In summary, we could say that WORD is used for variable declarations. It is worth mentioning that **assembler** is the name of the program that makes the mapping between identifiers and memory locations to ease the handling of data in assembly language. Now we can show you how to use WORD to work with integers instead of using binary numbers. In the following example, variables a, b, and c are declared, and their initial values are 9, 24, and 0 respectively.

Label	Operation	Initial value
a	WORD	9
b	WORD	24
c	WORD	0

We can now write a program to add a couple of numbers, say 24 and 9, and forget about memory addresses.

Program 1

Line number	Label	Operation	Operands (or data values)	
1	a	WORD	9	DATA (data segment)
2	b	WORD	24	
3	c	WORD	0	
4		LOD	R[0], 0, a	TEXT (program segment)
5		LOD	R[2], 0, b	
6		ADD	R[3], R[1], R[2]	
7		STO	R[3], 0, c	

Now that you are familiar with some VM/0 assembly instruction and data declarations, we could introduce some special instructions to carry out input/output (I/O) operations. These special instructions known as **supervisor calls (SVC)**, or **system calls**, allow a user program to request service from the **operating system (OS)**. The only thing we need to know about the OS at this moment is that the operating system is a program that manages computer system resources efficiently and provides an environment for programs to run. As I/O operations are handled by the operating system, when a program needs to input or output data, a call must be made to the OS, and the operating system will execute the I/O operation on its behalf. The way to write down these system calls in VM/0 assembly language is:

```
SYS 0, 0, 0    ; Program stops, waits for user input.
SYS 0, 0, 1    ; Displays a value on the monitor screen.
```

Although the way I/O is handled by the OS is hidden from the user program, we will give you an idea of how it occurs. In memory-mapped I/O (MMIO) there is, in the OS area, a memory location or I/O buffer associated with each device. VM/0 has only two I/O devices: a keyboard and a monitor. The keyboard is considered the **standard input (stdin)** and is identified with the number 0. The monitor is the **standard output (stdout)** and is identified with the number 1. The I/O buffer for the keyboard will be denoted as INBUF and the one for the monitor OUTBUF.

As for SYS 0, 0, 0 (input) this is what occurs:

1. The program stops momentarily and waits until the user types in a value on the keyboard and presses the return key.
2. The operating system moves the input value from the keyboard port to INBUF and then from INBUF to the register with a higher denomination; in VM/0 it would be R[3].

   R[3] ← INBUF ← Keyboard

Note: We will need an STO instruction right after SYS 0, 0, 0 to move the input value from the R[3] register to a memory location.

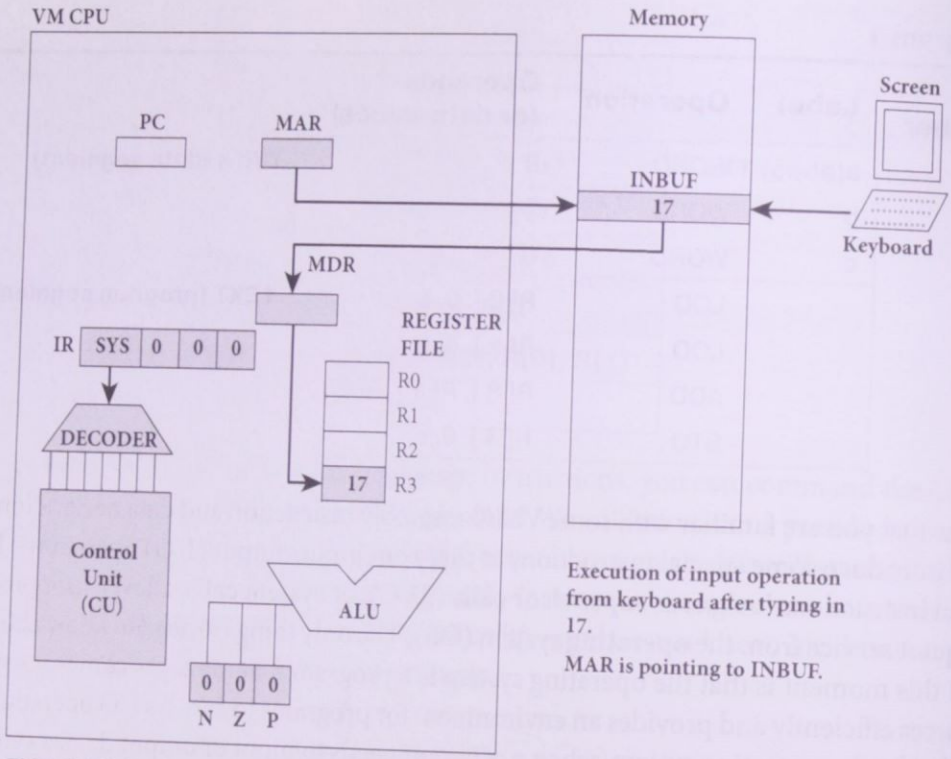


FIGURE 2.8 Input operation.

As for **SYS 0, 0, 1 (output)** this is what occurs:

1. The operating system stores the content of R[3] into the OUTBUF and then sends the contents of the OUTBUF to the monitor port.
2. The monitor displays the content of the monitor port on the screen.

Note: We will need an LOD instruction before SYS 0, 0, 1 to move a data value from a memory location to the R[3] register.

These two instructions used back to back allow the program to accept data from the keyboard and display it on the screen (echo). Figures 2.8 and 2.9 depict the input/output devices connected to memory.

Another system call you need to know at this point is **end of program (EOP)**. This system call tells the operating system that the program has run to completion properly. The operating system will take the appropriate actions to collect the resources the program was using.

```
SYS 0, 0, 2    ; End of program (EOP)
```

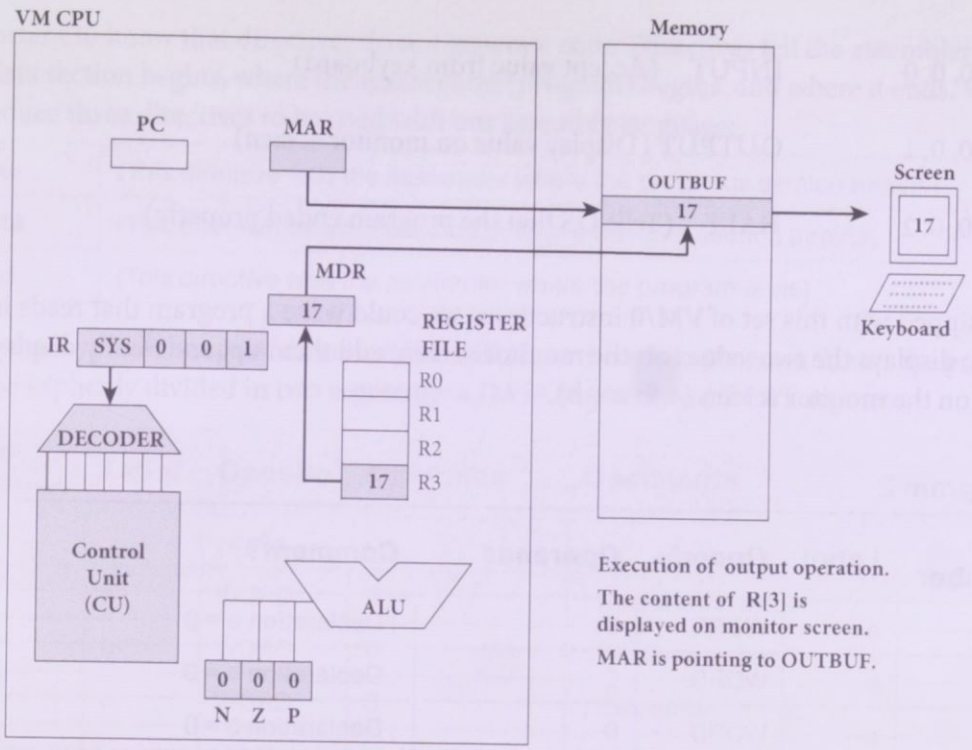


FIGURE 2.9 Output operation.

Let us proceed to make a list of all VM/0 instructions we have described thus far. In this list we will use x, y, and z to refer to register numbers, and they can take any value from zero to three. M is a modifier that has different interpretations depending on the instruction:

Mnemonic	Meaning	
LOD Rx, 0, M	Rx ← Mem [M]	(Copy the content of memory location M into Rx)
STO Rx, 0, M	Mem [M] ← Rx	(Copy the content of Rx into memory location M)
ADD Rx, Ry, Rz	Rx ← Ry + Rz	
SUB Rx, Ry, Rz	Rx ← Ry – Rz	
EQ Rx, Ry, 0	Set Z to 1 if Rx = Ry	
JPC 0, 0, M	Set PC to M (PC ← M), if Z = 1 or N = 1 or P = 1 (here M is an address)	
JMP 0, 0, M	Set PC to M (PC ← M)	

SYS 0, 0, 0	INPUT (Accept value from keyboard)
SYS 0, 0, 1	OUTPUT (Display value on monitor screen)
SYS 0, 0, 2	HALT (Tells OS that the program ended properly)

Equipped with this set of VM/0 instructions, we could write a program that reads in two values, displays the two values on the monitor screen, add them up, and finally displays the result on the monitor screen (c ← a + b).

Program 2

Line number	Label	Opcode	Operands	Comments
1	a	WORD	0	; Declaration a = 0
2	b	WORD	0	; Declaration b = 0
3	c	WORD	0	; Declaration c = 0
4		SYS	0, 0, 0	; R[3] ← keyboard
5		STO	R[3], 0, a	; a ← R[3]
6		SYS	0, 0, 1	; Display R[3] content on monitor
7		SYS	0, 0, 0	; Input from keyboard into R[3]
8		STO	R[3], 0, b	; b ← R[3]
9		SYS	0, 0, 1	; Display R[3] content on monitor
10		LOD	R[0], 0, a	; R[0] ← a
11		LOD	R[1], 0, b	; R[1]← b
12		ADD	R[0], R[0], R[1]	; R[0] ← R[0] + R[1]
13		STO	R[0], 0, c	; c ← R[0]
14		LOD	R[3], c	; R[3] ← c
15		SYS	0, 0, 1	; Display R[3] content on monitor
16		SYS	0, 0, 2	HALT: Tells OS that the program ended

Assembler Directives

The next step to improve VM/0 assembly language is the incorporation of **assembler directives** to assist the assembler in translating the program into binary (executable code). It is

important to know that directives do not generate code. Directives tell the assembler where the data section begins, where the text section (program) begins, and where it ends. We will introduce three directives to be used with our assembly language:

.text	(This directive tells the assembler where the programs section begins)
.data	(This directive tells the assembler where the data section begins)
.end	(This directive tells the assembler where the program ends)

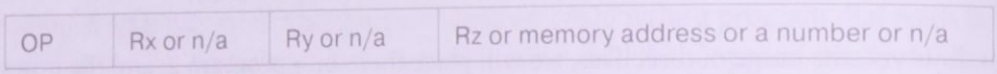
To show you the usage of directives, we will place them into program 2, and the program will be explicitly divided in two segments: a DATA segment and TEXT segment.

Line number	Label	Opcode	Operands	Comments
1		**.data**		
2	a	WORD	0	; Declaration a = 0
3	b	WORD	0	; Declaration b = 0
4	c	WORD	0	; Declaration c = 0
5		**.text**	main	; Program begins at main (line 6)
6	main	SYS	0, 0, 0	; R[3] ← INBUF ← keyboard
7		STO	R[3], 0, a	; a ← R[3]
8		SYS	0, 0, 1	; Display (R[3] → OUTBUF → Screen)
9		SYS	0, 0, 0	; Input from keyboard into IOB
10		STO	R[3], 0, b	; b ← R[3]
11		SYS	0, 0, 1	; Display (R[3] → OUTBUF → Screen)
12		LOD	R[0], 0, a	; R[0] ← a
13		LOD	R[1], 0, b	; R[1] ← b
14		ADD	R[0], R[0], R[1]	; R[0] ← R[0] + R[1]
15		STO	R[0], 0, c	; c ← R[0]
16		LOD	R[3], c	; R[3] ← c
17		SYS	0, 0, 1	; Display (R[3] → OUTBUF → Screen)
18		SYS	0, 0, 3	; HALT: Tells OS that program ended
19		**.end**		

Note: Label main indicates where the program begins.

Instruction Set Architecture

The **instruction format** of VM/0 has four components, which we will refer to as fields. The CPU will handle each field differently depending on the opcode. The ISA of the VM/0 has 20 instructions divided in two groups. Instructions beginning with opcode 0000 are used for arithmetic and relational operations, and all other instruction have an opcode not equal to 0000. We will show how they are encoded using the following instruction format.



VM/0 Instruction Format

Where,

OP Is the operation code (for example: LOD)
Rx Unused or refers to a register in the register file (**destination register**)
Ry Unused or refers to a register in the register file (**source register**)
Rz Depending of the opcode it indicates:
A register in arithmetic instructions (**source register**)
(e.g. ADD R[1], R[2], R[3])
A constant value, or literal (instruction: LIT)
A program address (instruction JMP)
A data address (instructions: LOD, STO)

There are 11 arithmetic/relational operations that manipulate the data within the register file. These operations will be explained after some basic VM/0 instructions. Rx, Ry, and Rz can be replaced by 0, 1, 2, and 3 depending what register of the register file you want to use.

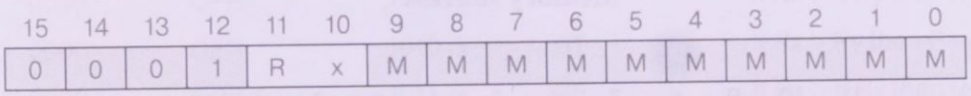
(Opcode)		
01 – **LIT**	**Rx, 0, M**	Load a constant value (literal) **M** into register **Rx**
02 – **LOD**	**Rx, 0, M**	Load value into Rx from memory location **M**
03 – **STO**	**Rx, 0, M**	Store value from Rx into memory location **M**
04 – **JMP**	**0, 0, M**	Jump unconditionally to memory location **M (PC ← M)**
05 – **JPC**	**0, 0, M**	Jump on condition: if any CC equals 1, then PC ← **M**
15 – **SYS**	**0, 0, 0**	Read in input from keyboard and store it in register R3
SYS	**0, 0, 1**	Display content of register R3 to the screen
SYS	**0, 0, 2**	End of program (program stops running)

In the table below we will present the opcode in binary of the instructions explained so far. All arithmetic operations have the same opcode **OPR** (OPR = 0000) and a function code to select the appropriate arithmetic operation, as shown below.

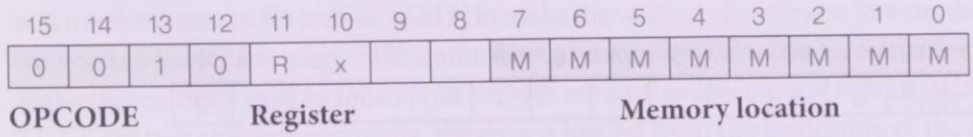
Opcode in binary	Mnemonic	Function in binary	Arithmetic/relational operation
0000	OPR	0001 →	ADD
0000	OPR	0010 →	SUB
0000	OPR	0110 →	EQL
0001	LIT		
0010	LOD		
0011	STO		
0100	JMP		
0101	JPC		
1111	SYS		

Now let us take a look at the encoding of these instructions in 16-bit words:

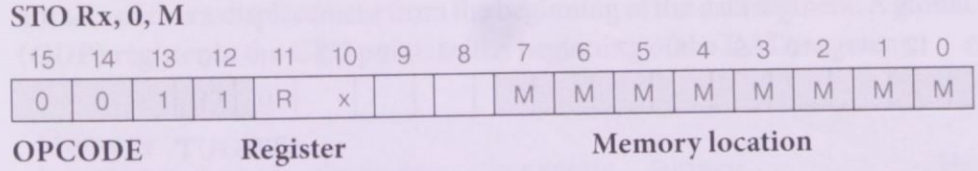
LIT Rx, 0, M



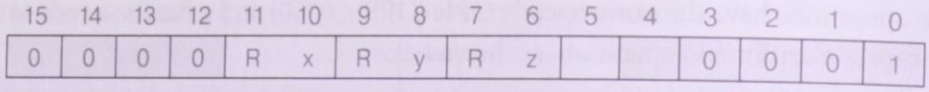
LOD Rx, 0, M



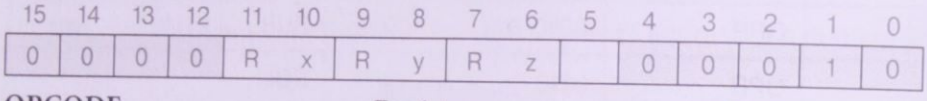
STO Rx, 0, M



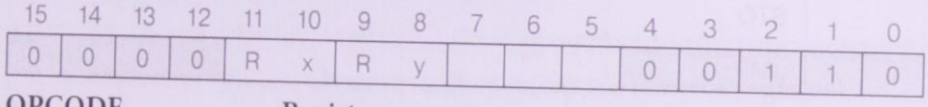
ADD Rx, Ry, Rz



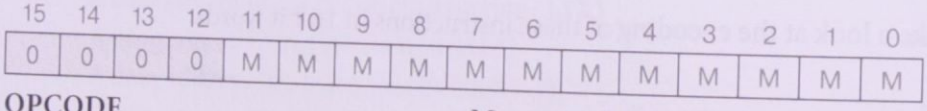
SUB Rx, Ry, Rz



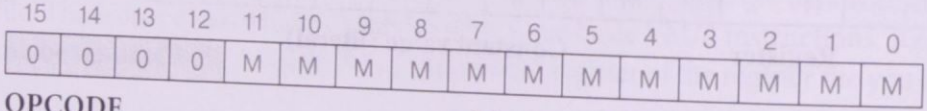
EQ Rx, Ry, 0



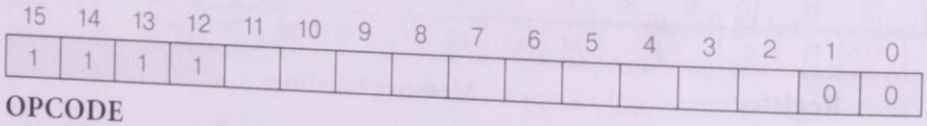
JPC 0, 0, M



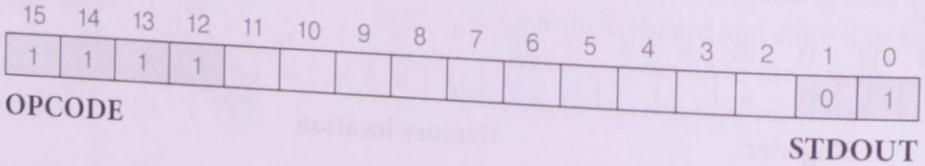
JMP 0, 0, M



SYS 0, 0, 0



SYS 0, 0, 1



SYS 0, 0, 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1											1	0
OPCODE														**EOP**	

The function values for all arithmetic and relational operations and the description of the actions taken by each instruction are shown below:

Opcode in binary	Mnemonic	Function in binary	Arithmetic/ relational operation	Description
0000	OPR	0001 →	ADD	Rx ← Ry + Rz
0000	OPR	0010 →	SUB	Rx ← Ry − Rz
0000	OPR	0011 →	MUL	Rx ← Ry * Rz
0000	OPR	0100 →	DIV	Rx ← Ry / Rz
0000	OPR	0101 →	MOD	Rx ← Ry mod Rz
0000	OPR	0110 →	EQ	If Rx = = Ry then Z = 1
0000	OPR	0111 →	NE	If Rx ! = Ry then Z = 0
0000	OPR	1000 →	LT	If Rx < Ry then N = 1
0000	OPR	1001 →	LE	If Rx = < Ry then N = 1 or Z = 1
0000	OPR	1010 →	GT	If Rx > Ry then P = 1
0000	OPR	1011 →	GE	If Rx > = Ry then P = 1 or Z = 1

Now that you have a better understanding of the VM/0 instruction set architecture, let us take a look at program 2 translated into binary and hexadecimal. By the way, from now on we will use the notation R3 instead of R[3] to make the writing of programs less cumbersome. Also, we need to justify the usage of line numbers as memory addresses for variables a, b, and c to make the example easy to follow, but we will repair this inaccuracy right now. As DATA and TEXT are two separate segments, the data is loaded from the beginning of the data segment and the instructions are loaded at the beginning in the TEXT segment. All variable addresses are accessed as a displacement from the beginning of the data segment. A **global data pointer (GDP)** register in the CPU points to the beginning of the DATA segment.

Memory location	Label	Opcode	Operands	Binary	Hexadecimal
		.data		; no code generated	
0x0	a	WORD	0	0000000000000000	0x0000 ← GDP

0x1	b	WORD	0	0000000000000000	0x0000
0x2	c	WORD	0	0000000000000000	0x0000
		.text main		**; no code generated**	
0x0	main	SYS	0, 0, 0	1111000000000000	0xF000 ← PC
0x1		STO	R3, 0, a	0011110000000000	0x3C00
0x2		SYS	0, 0, 1	1111000000000001	0xF001
0x3		SYS	0, 0, 0	1111000000000000	0xF000
0x4		STO	R3, 0, b	0011110000000001	0x3C01
0x5		SYS	0, 0, 1	1111000000000001	0xF001
0x6		LDO	R0, 0, a	0010000000000000	0x2000
0x7		LDO	R1, 0, b	0010010000000001	0x2401
0x8		ADD	R0, R0, R1	0000000001000001	0x0041
0x9		STO	R0, 0, c	0011000000000010	0x3002
0xA		LOD	R3, 0, c	0010110000000010	0x2C02
0xB		SYS	0, 0, 1	1111000000000001	0xF001
0xC		SYS	0, 0, 3	1111000000000010	0xF002
		.end		**; no code generated**	

Process Address Space

Once an assembler translates a program into binary, we refer to it as **object code**. In order to run the object code, we need to load it into memory. When the program is running, we call it a **process**; hence, we define a process as a **program in execution**. To run a program, the operating system must assign to the program a memory area called the **process address space** (or user address space). This process address space is divided into four segments: **stack segment**, **heap segment**, **data segment**, and **text segment**. In this chapter, we have been dealing with the text segment and data segment. The stack segment, which will be discussed in chapter 3, is used to manage the calling and returning from subroutines. The heap segment, which will be discussed in chapter 4, is used to provide memory on demand to allow some programming languages to handle memory dynamically. Figure 2.10 depicts the process address space; you can observe that the text segment begins at location 0x0000, followed by the data segment beginning at memory location 0x2000. You will find the heap segment starting at location 0x4000 and the stack segment beginning at location 0x6000. The user address space ends at location 0x7FFF. The operating system area or kernel address space uses all memory locations from 0x8000 to 0xFFFF. The size of every segment in the user address space is based on the ones proposed by Knuth in his MMIX computer.

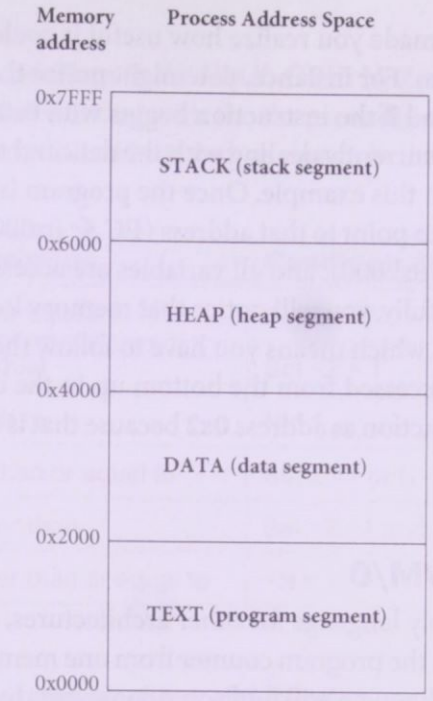


FIGURE 2.10 Process address space.

We are now ready to load the assembled object code for program 2 into the process address space. We will use hexadecimal next to each line in TEXT and variable names next to each line in DATA to make the understanding of the object code easier.

Memory address	Process Address space		
	0000 0000 0000 0000	c	**DATA (data segment)**
	0000 0000 0000 0000	b	
GDP → 0x2000	0000 0000 0000 0000	a	
	1111 0000 0000 0010	0xF002	**TEXT (program segment)**
	1111 0000 0000 0001	0xF001	
	0010 1100 0000 0010	0x2C02	
	0011 0000 0100 0010	0x3002	
	0000 0000 0100 0001	0x0041	
	0010 0100 0100 0001	0x2401	
	0010 0000 0100 0000	0x2000	
	1111 0000 0000 0001	0xF001	
	0011 1100 0100 0001	0x3C01	
	1111 0000 0000 0000	0xF000	
	1111 0000 0000 0001	0xF001	
	0011 1100 0000 0000	0x3C00	
PC → 0x0000	1111 0000 0000 0000	0xF000	

This example might have made you realize how useful it could be to have a hexadecimal representation of the program. For instance, you might notice that each instruction starting with a 0xF is a system call, and if the instruction begins with 0x0, it is an arithmetic or relational instruction. As we are currently dealing with the data and text segments, the stack and heap segments are omitted in this example. Once the program is loaded in the text segment at location 0x0, the PC is set to point to that address (PC ← 0x0000). The global data pointer is set up to 0x2000 (GDP ← 0x2000), and all variables are accessed as a displacement from the GDP. If you observe carefully, you will notice that memory location zero is at the bottom of the process address space, which means you have to follow the program from the bottom up. Similarly, variables are accessed from the bottom up in the data segment. For instance, variable **c** is encoded in instruction as address **0x2** because that is the distance (displacement) from the GDP.

Some Remarks on VM/0

When reading about assembly language for other architectures, you will find that the PC is incremented by four to move the program counter from one memory address to the next, and the word length is 32 bits. Below, you will find some comments to help you better understand the assembly language of architectures like MIPS, Intel, and ARM.

On Word Length and Byte Addressable Memories

VM/0 word length is 16 bits, which means that to move from one memory location to the next, a jump of 16 bits must be done. We will use the VM/0 fetch cycle presented in section 2.3 to explain:

```
FETCH: MAR ← PC
       PC  ← PC + 1
       MDR ← MEM[MAR]
       IR  ← MDR
```

As you can see, the PC is incremented by one because VM/0 is word addressable. There are current architectures like MIPS and ARM that are byte addressable, which means they can move from one byte to the next. Then if we have a 32-bit word and memory is byte addressable, the PC must be incremented by four to move from one word to the next. Hence, FETCH will be executed this way:

```
FETCH: MAR ← PC
       PC  ← PC + 4
       MDR ← MEM[MAR]
       IR  ← MDR
```

If VM/0 were byte addressable, the PC would have to be incremented by two in the Fetch cycle.

On Condition Codes

There are three condition code flags in the VM/0-CPU: N, Z, and P. We included the P flag because it was more intuitive working with the three conditions separately. However, only with flags N and Z can all conditions be tested, as shown in the following table.

Description	Condition Codes
Test for equality	Set Z = 1
Not equal	Set ~Z = 1
Less than	N = 1
Less than or equal to	Set Z =1 or N = 1
Greater than	Set ~Z =1 and ~N = 1
Greater than or equal to	~N = 1

As you know, VM/0 is a virtual machine, and therefore we do not need to implement all the condition codes of an actual processor. In CPUs such as MIPS or Intel, there are other flags to detect overflow (V flag) or carry (C flag).

On Increasing VM/0 Word Length

We have described VM/0 using a 16-bit word to keep it simple. However, all concepts presented so far can be implemented in a 32- or 64-bit word. For instance, using 32 bits we can do the following:

a. Use 6 bits for the opcode, which will allow us to represent $2^6 = 64$ different opcodes.
b. Use 4 bits to represent registers and increase the numbers of registers in the register file to 16.

SUMMARY

In this chapter, we reviewed the concepts of positional number systems and highlighted the importance of binary numbers because that is the language computers use. Then we introduced an assembly language (the ISA) for a virtual machine (VM/0) and explained how each instruction was executed. This ISA allowed us to write simple programs for VM/0. Then we introduce a system program called the assembler, which translates assembly language into binary to allow the CPU to execute the user program. Finally, we introduced the concept of process as a program in execution and explained that each time we need to run a program, the operating system creates a process address space for the program.

EXERCISES

1. Write down the numbers from 0 to 20 in decimal, binary (use 8 bits), and hexadecimal.
2. Convert the followings numbers to decimal:

   a) 00010001 b) 10010111
   c) 0xFA d) 0xABC

   Hint: Use $\sum_{k=0}^{n-1} d_i b^i$ where **b** stands for base and **d** stands for digit. For example, in 10101010_2, n = 8 and the b = 2:

1	0	1	0	1	0	1	0
↓	↓	↓	↓	↓	↓	↓	↓
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
×	×	×	×	×	×	×	×
b^7	b^6	b^5	b^4	b^3	b^2	b^1	b^0

3. Convert the following binary numbers into hexadecimal:

   a) 1111000000000001 b) 0000000001000001
   c) 0000110100000110 d) 0000100111000011

4. Convert the following hexadecimal numbers into binary:

   a) 0x0D41 b) 0xF001
   c) 0x200A d) 0x3C03

5. The binary strings in question 3 represent the encoding of four VM/0 instructions. Convert the strings into assembly language.
6. Write down the steps carried out in VM/0 fetch cycle assuming memory is byte addressable.
7. What operation executes the VM/0 control unit to find out which one of the condition codes might be set to one?
8. Write down the steps the VM/0 control unit must execute for the instructions:

   a) STO R2, 0, 10
   b) LOD R3, 0, 5
   c) JMP 0, 0, 7

Bibliographical Notes

A tiny p-code interpreter was proposed by Wirth[1] as the target machine to implement the PL/0 compiler. MIXAL assembly language for a hypothetical computer was proposed by Knuth[2] in his book series *The Art of Computer Programming* as a tool to teach algorithm, data structures, and programming. Wirth's RISC interpreter[3] was proposed as a virtual machine to be used as the target machine for the Oberon Compiler. MMIX, a RISC hypothetical computer, and its assembly language[4] were proposed to replace MIX architecture. A good introduction to Intel x86 CPU is presented by Bryant and O'Hallaron,[5] and for an in-depth understanding of MIPS processors, we refer readers to Patterson and Hennessy.[6] There is also an excellent description of the ARM architecture presented by Patterson and Hennessy.[7] About the origins of binary numbers and binary arithmetic, we refer readers to Leibnitz.[8]

Bibliography

1. N. Wirth, *Algorithm + Data Structures = Programs*. Upper Saddle River, NJ: Prentice Hall, 1976.
2. D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, 3rd ed., vol. 1. Upper Saddle River, NJ: Addison-Wesley, 1973.
3. N. Wirth, *Compiler Constructions*. Redwood City, CA: Addison-Wesley, 1996.
4. D. E. Knuth, *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX a Risc Computer for the NEW Millennium*. Upper Saddle River, NJ: Addison-Wesley, 2005.
5. R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Boston, MA: Prentice Hall, 2016.
6. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Cambridge, MA: Morgan Kaufmann, 2013.
7. D. A. Petterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface ARM Edition*. Cambridge, MA: Morgan Kaufmann, 2017.
8. G. G. Leibnitz, "Explication de l'arithmétique binaire," *Mémoires de mathématique et de physique de l'Académie royale des sciences* (1703), pp. 85–89.

Stacks, Recursion, and Nested Programs

So the last shall be first, and the first last.
Matthew 20:16
—The Bible

INTRODUCTION

The stack principle, or last-in-first-out storage method, was initially developed to evaluate expressions written using the parenthesis-free notation known as reverse Polish notation. The beauty of this method attracted the attention of many researchers, and almost immediately the stack concept was used to implement recursive procedures in programming languages, developing compilers, and building stack organized computers. Nowadays, the stack principle is used in computer systems mainly for subroutine linkage.

We initiate this chapter with a detailed explanation of the stack mechanism, or pushdown storage. Then glancing backward to the process address space, we remind you that the base of the stack is at the top of the process address space, and therefore the stack pointer has to be decremented by one for the stack to grow and incremented by one to shrink. Then we present the concept of activation record as a means for subroutine linkage. Finally, the implementation of recursion and nested procedures is described.

CHAPTER OBJECTIVES

- To review the concept of pushdown storage, or stack mechanism.
- To discuss the usage of the stack and activation records to control subroutine linkage.
- To explore the implementation of recursive functions using several instances of the same activation record.
- To understand the implementation of nested procedures.

VOCABULARY

This is a list of keywords that herald the concepts we will study in this chapter. They are organized in chronological order and will appear in the text in bold. We invite you to take a look at these keywords to find out which ones you are familiar with.

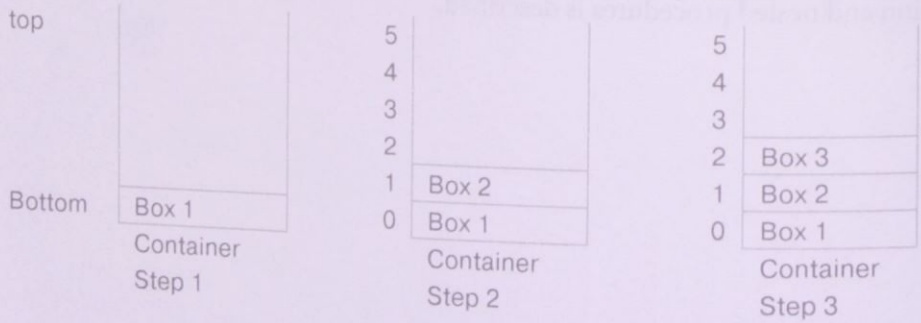
Last in first out (LIFO)
SP
Push (data value)
Pop ( )
Text
Data
Heap
Stack (run-time stack)
Activation record (AR)
Stack frame
Base pointer (BP)
Stack pointer (SP)
Functional value (FV)
Dynamic link (DL)
Return address (RA)
Parameters
Local variables
Increment (INC)
Call (CAL)
Return (RTN)
Reserved word
PL/0
Var
Statement
Assignment statement
Constant
Main
Begin
End
Subroutine declaration
Procedure
Function
If-statement
If-then
If-then-else
While-do
Recursion
Lexicographical level
Static link (SL)
Static chain

ACTIVATING PRIOR KNOWLEDGE

In this section we will present a series of activities. In some of them you can choose one or more options. Sometimes, if you do not agree with the given answers to choose from, you will be allowed to give your own answer. By the way, this is not a test.

1. Is there anything that looks familiar to you in these three options?
   a. 6
   b. 3!
   c. $3 \times 2 \times 1$

2. Assume you have three boxes and a container. The boxes are placed in the container as shown below: First box 1, then box 2, and finally box 3.



If placing a box into the container is considered a push step and getting a box out of the container is considered a pop step and boxes can only move in and out through the container top:

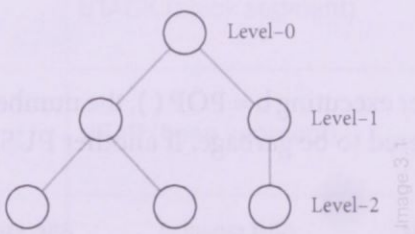
a. Find out how many push steps are necessary to put box 1 inside the container.
b. Find out how many pop steps are necessary to get box 1 out of the container in step 3.

3. Skim the chapter and pay attention to the words written in **bold**. Count the number of words you are familiar with.

4. Does LIFO = FILO?

5. The following expression 3 4 * 5 6 * + is expressed in reverse Polish notation. Rewrite this expression using infix notation.

6. Observe the following picture. Each level in the three is called a lexicographical level, and the root is level zero:



a. How many steps are necessary to move from level 2 to level 0?
b. How many steps are necessary to move from level 1 to level 0?
c. How many steps are necessary to move from level 2 to level 1?

Stack Mechanism

The stack is a data structure that could be implemented using an array, and its management follows the **last in first out (LIFO)** behavior. In the CPU there is a register called the stack pointer (**SP**), which points to the top of the stack, and two operations are used to insert data values into or remove them from the top of the stack.

1. To place a value on top of the stack, we can use the operation push, whose format is: **PUSH (*data value*)**. This operation increments the SP by one and inserts a data value on top of the stack. These are the steps for the PUSH operation:

PUSH (a), means → SP = SP + 1
Stack [SP] = *a*

2. To remove a value from top of the stack and store it in a variable, say b, we can operation **POP( )**. This operation removes whatever data value is found on to stack and decrements the stack pointer by one, as described below:

b = **POP ( )**, means → *b* = Stack [SP]
SP = SP - 1

An illustration will help illustrate the effects of these operations. In figure 3.1 the t shows three stack states: In the initial state, the SP points to stack location zero, whose c is the value 7. Then we show the stack after executing PUSH (23), and finally on the hand side, we present the stack after executing PUSH (29). It is worth mentioning these stages, the values in the shaded area are unreachable with a single application ( ). In the second row of stacks in figure 3.1, observe the three stack states after apply following operations to the stack in the right-hand side in the top row:

b = POP ( )
PUSH (5)
PUSH (31)

It is worth noticing that after executing b = POP ( ), the number 29 is in stack locatio is unreachable and is considered to be garbage. If another PUSH operation is execut

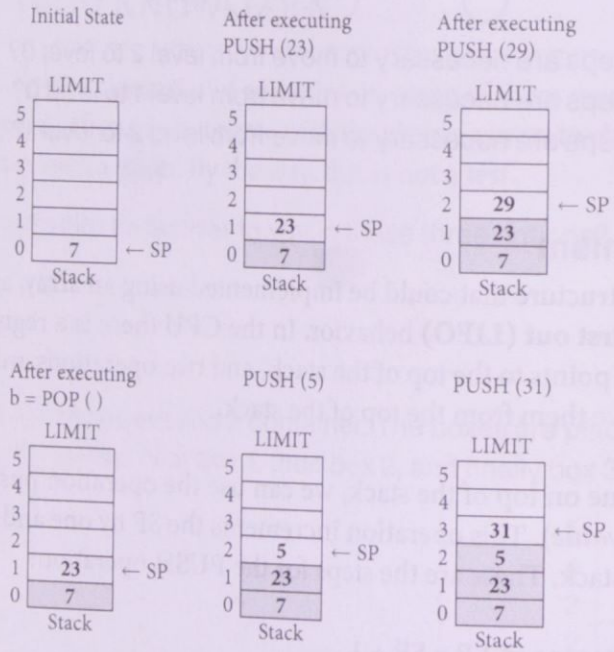


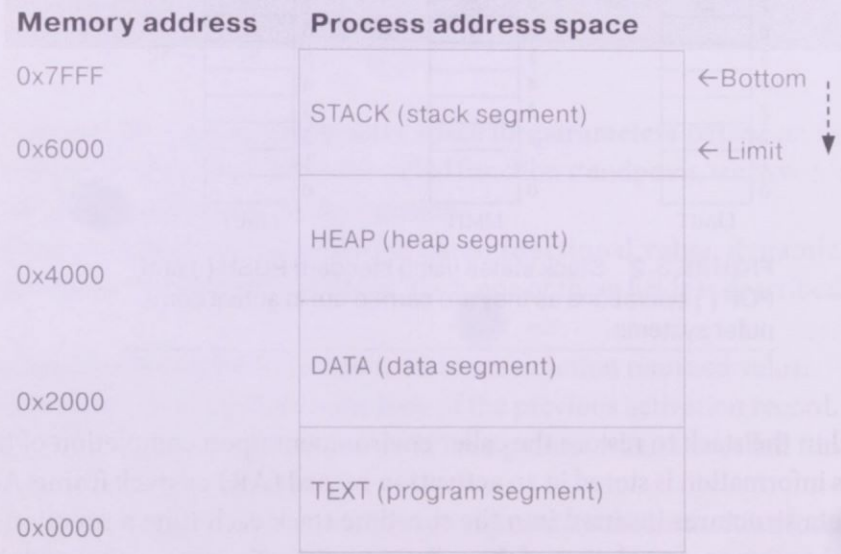
FIGURE 3.1 Stack states after using PUSH ( ) and POP ( ) operations.

example PUSH (5), the 29 will be overwritten with the value 5. The stack state after executing PUSH (31) can be seen in the right-hand side of the second row of stacks.

VM/0 Stack Architecture

VM/0 is a stack machine that provides a process address space for programs to run. Recall from chapter 2 that the process address space is a memory area organized in four segments: the **text** or code segment, which contains a list of instructions; the **data** segment, which stores global variables declared in the main program; the **heap**, which provides memory to the program on demand; and the **stack (run-time stack)**, which stores local variables and manages the changing of environments by keeping track of subroutine calls and returns.

Let us bring back an image of the process address space as a reminder that the stack is placed at the top of the address space.



As the stack bottom is located at the top of the process address space, the stack grows downward and shrinks upward. Therefore, we must redefine the way PUSH and POP work in the VM/0 computer system as follows:

PUSH (data value)	b = **POP ( )**
SP = SP – 1	b = Stack [SP]
Stack [SP] = *a*	SP = SP + 1

Figure 3.2 shows the example illustrated in figure 3.1 using the new version of the PUSH and POP operations.

When a program is running, it frequently changes from the execution of one function to another. Each time a function is called, some information from the caller and called function

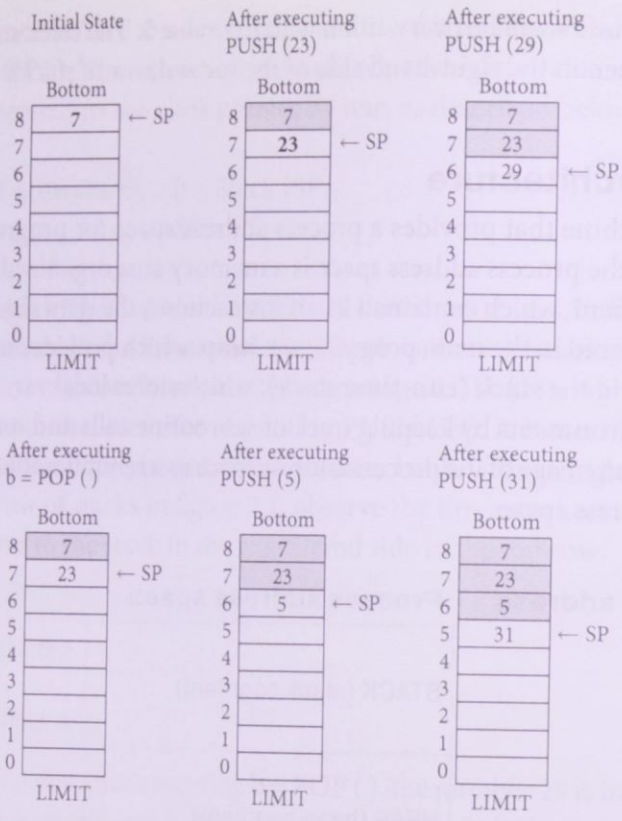


FIGURE 3.2 Stack states using standard PUSH ( ) and POP ( ) operations as they are carried out in actual computer systems.

must be saved in the stack to restore the caller environment upon completion of the called function. This information is stored in an **activation record (AR)** or **stack frame**. Activation records are data structures inserted into the run-time stack each time a function is called in and popped out upon completion of the called function. To manage the stack in VM/0, we need to add three more registers to the VM/0 CPU. These registers are the global data pointer (GP), which points to the data area; the **base pointer (BP)**, which points to the base of the current activation record; and the **stack pointer (SP)**, which points to the top of the stack. Figure 3.3 depicts the BP and SP pointing to an AR in the stack and the GP pointing to the data area.

Note: The words *procedure*, *function*, *method*, and *subroutine* will be used interchangeably.

Activation Records and Procedure Calls

There is an activation record associated with each function or procedure, and each time a function is called, its activation record is inserted on the stack. The data stored in the AR is a mix of caller and called function information. From the caller we need to store the contents

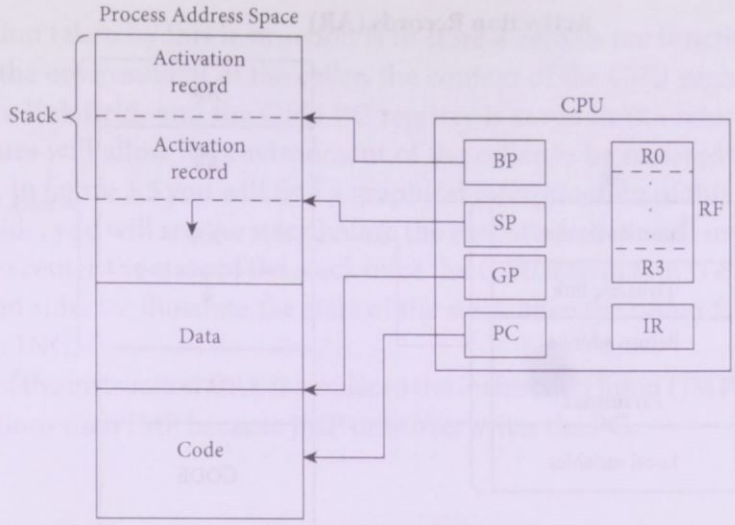


FIGURE 3.3 The GP, SP, and BP pointing to the stack in the process address space.

of the PC register, BP register, and reserve space for **parameters** passed on from the caller function to the called function. From the called function standpoint, we have to reserve space for the **local variables** declared in the function.

The AR layout consists of the following parts: **functional value**, **dynamic link**, **return address**, **parameters**, and **local variables**. Each one of these fields is described as follows:

- **Functional value (FV):** Location to store the function returned value.
- **Dynamic link (DL):** Points to the base of the previous activation record. This address is copied back from the DL to the CPU base pointer when the called function ends to restore the caller environment.
- **Return address (RA):** Points, in the code segment, to the instruction located right after the **call** instruction executed by the caller. This address is copied from the RA to the CPU program counter after termination of the current function or procedure (called function) to restore the caller environment.
- **Parameters:** Space reserved to store the actual parameters the caller function is passing on to the called function.
- **Local variables:** Space reserved to store local variables declared within the called function.

Let us explore and find out where the above mentioned information is stored in the AR. Figure 3.4 presents the data structure of called AR and shows where each field is placed.

To understand the way ARs are inserted on the stack, we need to add three new instructions to the VM/0 ISA. These new instructions will be used by a program to **increment (INC)** the stack pointer to accommodate the activation record, to **call (CAL)** a function, and to **return**

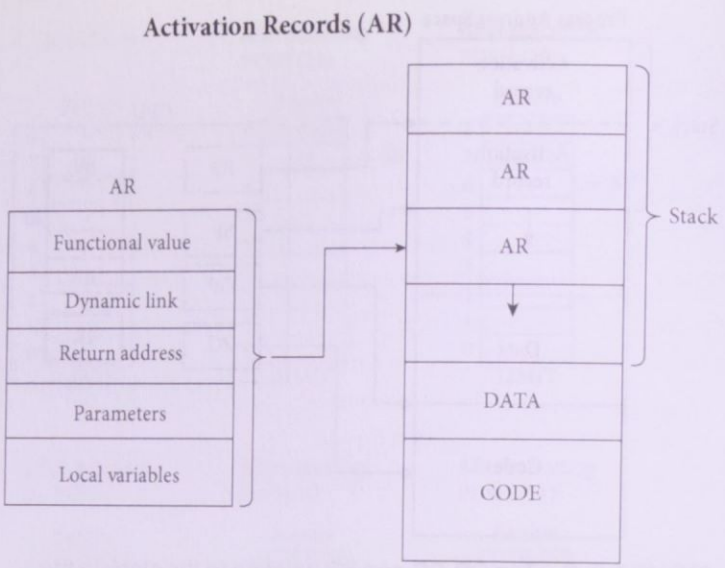


FIGURE 3.4 Activation record description and their placement in the stack.

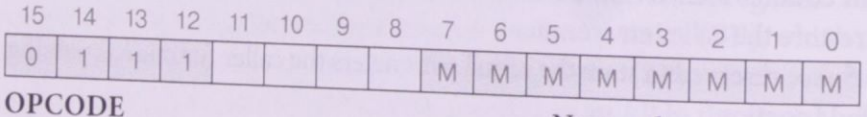
(RTN) from a subroutine to the caller function once the called function ends. These new instructions are described below:

(Opcode)

07 – **INC** 0, 0, **M** sp ← sp – **M**; Reserve space for AR

The instruction INC reserves space in the stack to place an AR each time a function is called. The **M** field in the instructions indicates the number of words to be reserved. This is carried out by subtracting the value **M** from the **SP (remember that the stack grows downward)**. The number of words reserved includes space for FV, DL, RA, parameters, and local variables. The instruction INC is always placed at the beginning of each function, including the main function, and its instruction format is given below:

INC 0, 0, M



The second instruction, CAL, is used to call a subroutine in a program. The description of the actions taken by this instruction is shown below:

(Opcode)

06 – **CAL** **0, 0, M**

stack[sp – 1] ← 0;	FV ← 0
stack[sp – 2] ← bp;	DL ← CPU.BP
stack[sp – 3] ← pc;	RA ← CPU.PC
bp ← sp – 1;	Set BP for callee
pc ← **M**;	Points to subroutine

The first action taken by this instruction is to store a zero in the functional value field. Then, to save the environment of the caller, the content of the CPU register BP is stored in the dynamic link field, and the CPU PC register is saved in the return address field. These two values will allow the environment of the caller to be restored when the called function ends. In figure 3.5 you will find a graphical interpretation of this instruction. On the left-hand side, you will see the stack before the execution of the call instruction (CAL) and then in the center the state of the stack once the CAL instruction is executed. Finally, on the left-hand side, we illustrate the state of the stack once the called function executes the instruction INC.

The format of the instruction CAL is similar to the instruction jump (JMP), but CAL takes many more actions than JMP because JMP only overwrites the PC.

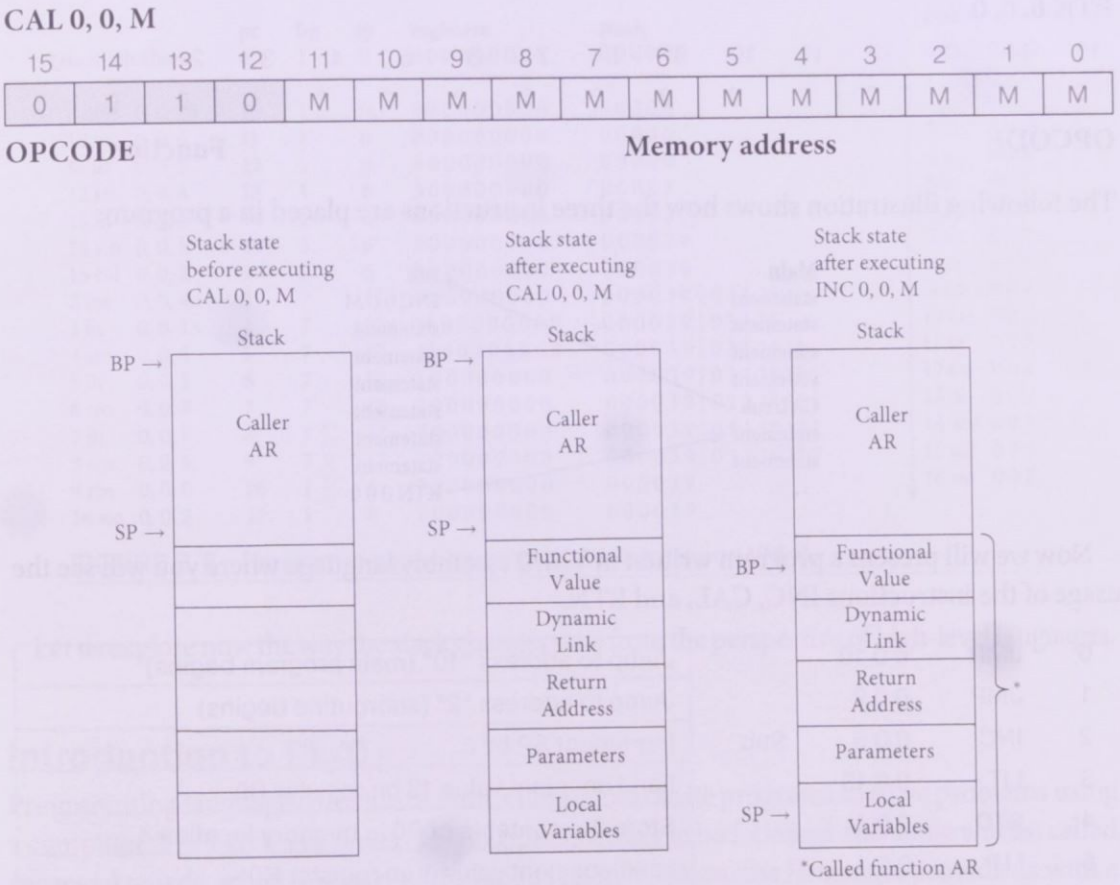


FIGURE 3.5 Insertion of an activation record in the stack.

The third instruction is return. Each time a subroutine ends, the caller must take control of the CPU. Placing the instruction return (RTN) as the last instruction of the subroutine will accomplish this task.

PL/0 is a small programming language proposed by Niklaus Wirth, and it could be considered a tiny subset of Pascal. We present here a variant of PL/0 with some added programming constructs that will help us explain some concepts. A program in PL/0 consists of two parts: declarations and statements. Each variable declaration reserves a memory location and associates a name (aka identifier) with that memory location. The way to declare a variable in PL/0 is by using the reserved word **var** follow by an identifier; for example, to declare the variables a, b, and c, we have to write:

var a, b, c;

Statements are used to tell the computer what has to be done. For instance, the **assignment statement**

a:= b + c;

instructs VM/0 to execute an arithmetic operation by adding the contents of variables b and c. The assignment symbol ":=" tells the computer to store the resulting addition value in variable a. The statement can be read this way: "a becomes b + c." You may have noticed that variable declarations and statements end with a semicolon (";") symbol to indicate the end of the declaration or the end of a statement. This is mandatory to comply with PL/0 syntax rules.

Sometimes you may be interested in declaring a **constant**. A constant is a value in a memory location that cannot be altered. For example:

```
const k = 7;        // constant declaration
a:= b + k;          // we are adding b + 7
```

If k is used in the right-hand side of the assignment statement, an error will occur. Therefore, you must not write statements such as

k: = a; or k:= a + b;

To write a program in PL/0, there are some syntax rules we have to stick to. A program must begin with the word **main**, followed by constant declarations and variable declarations, then the keyword **begin**, followed by statements, and the keyword **end** right after the last statement. The final touch is the symbol "." (period) following the reserved word **end** to indicate the end of the program. This is an example of a PL/0 program:

```
1 main
2 var a,b,c;
3 begin
4  b:= 2;
```

```
5  c:= 5;
6  a:= b + c;
7  end.
```

Sometimes, there is a list of statements that must be executed several times in different parts of a program, and a clever way to avoid this repetition of statements is to group all those statements together under a single name and use the name instead. This technique is known as **subroutine declaration**. In PL/0 this is carried out by using the keyword **procedure** or **function**, followed by a name (identifier). Subroutines or functions can also be used as an approach to structure a program as a combination of modules. Let us use an example to illustrate this concept. There is a line indicating that the top part contains declarations of variables and procedures and the bottom part contains the main program statements.

PROGRAM 3.1

```
main
var n;                  // global variable declaration

function sub1;          // subroutine  sub1 declaration
var a, b, p;            // subroutine local variable declarations
a:= b + x + p;          // subroutine statements begin here
end;                    // end of sub1

function sub2(y);       // subroutine sub2 declaration (with parameter)
var c;                  // subroutine local variable declaration
c:=2 + y;               // subroutine statements start here
sub1;                   // sub2 calls sub1
end;                    // end of sub2
-----------------------------------------------------------------
begin
  n := 5;               // main program statements start here
  sub2(n);              // main calls sub2
end.
```

Program 3.1 shows a program with two functions, named sub1 and sub2. Main calls sub2, and a parameter is passed on from main to sub2; then sub2 calls sub1. In figure 3.7 you can see the stack activity for this program. The left-hand side shows the state of the stack while running main. The one in the middle is a snapshot after calling sub2. The shaded area highlights the sub2 AR. The stack on the left-hand side shows the stack once sub2 has called sub1 and the AR for sub1 was inserted into the stack.

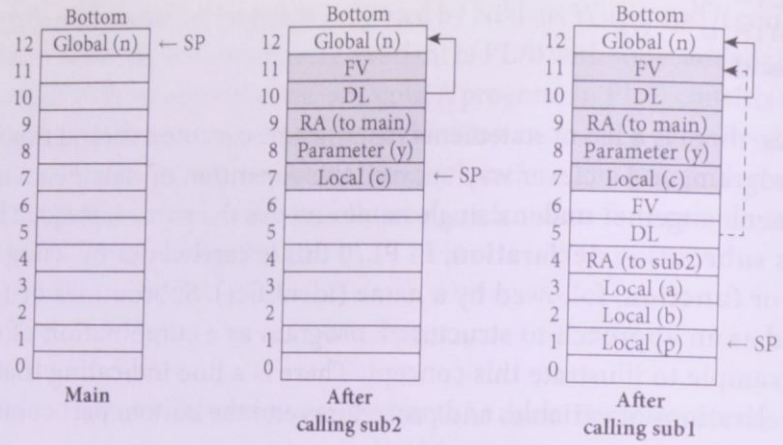


FIGURE 3.7 Stack states when program 3.1 is running.

Sometimes when writing a program, we come to a point that a decision has to be mad In PL/0 as in other programming languages, the programming construct to use is the **statement**. There are two variants for the if-statement:

1. **If** *condition* **then** *statement*;
   Example: **If** a > b **then** a := a + 1;

2. **If** *condition* **then** *statement* **else** *statement*;
   Example: **If** a > b **then** a := a + 1 **else** a := 0;

Iterations, also known as loops, are necessary in programming languages to execute a serie of statements while a condition holds. Once the condition is not met, the program gets out the loop. In PL/0 the programming construct to carry out iterations is the **while-statemen** The syntax you must use to write a while-statement is

while *condition* **do** *statement*;
Example: **while** a > 5 **do** a := a – 1;

You may have noticed that in both if-statements and while-statements, the syntax rul shows that the word *statement* follows the reserved words **then**, **else**, and **do**. Perhaps thi makes you think that only one statement can be placed after these reserved words, but tha is not the case. There are two variants of statements you can choose from: single statement and compound statements.

1. In single statements only one statement can be used.
   If a > b **then** a := a + 1;
   while a > 5 **do** a := a – 1;

2. In compound statements several statements can be used, but they must be enclosed between the reserved words **begin** and **end**.

```
If a > b then begin
        a := a + 1;
        c := d * e;
        end;
```

Recursive Programs

Quicksort is a fast, recursive sorting algorithm invented by C.A.R. Hoare in 1959. The algorithm was published in 1960. In 1980 Hoare received the Turing Award, and in his Turing lecture, "The Emperor's Old Clothes," he makes a statement acknowledging the relevance of recursion in programming languages:

> Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world.

Recursion is a beautiful and effective principle to handle complex problems. The way it works is by defining the solution of a problem in terms of a simpler version of itself. From a programming standpoint, it can be seen as a powerful method to write concise programs to handle complex problems; it consists of invoking a function that calls on itself. A recursive algorithm has two steps:

1. Explicit condition or base case (to stop the function from calling itself again)
2. The recursive call (where the function calls itself with a simpler version of the problem). We could consider a simpler version of the problem as a smaller data set.

Factorial is a good example to explain recursion. For example, factorial of 5 (5!) is defined as:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

The following PL/0 program is a recursive program to compute factorial. We call function fact with parameter n = 5, say fact(5). The function will verify if n = 1 returns the value 1; otherwise, it computes f = 5 * fact (4). As you see, we are calling function factorial again, but this time with parameter n = 4. The function verifies if n equals 1, but as that is not the case, fact(4) is replaced by 4 * fact(3). This means that f = 5 * fact(4) = 5 * 4 * fact(3). You can continue applying the same reasoning time and again until fact(1) is invoked, then the function will stop calling itself and you finally get:

$$Fact(5) = 5 * 4 * 3 * 2 * 1$$

Now the computation can be carried out, and you will obtain the answer 120.

```
program Factorial;
var f, n;

function fact(n): integer;
begin
 if n = 1 then
    return := 1;
 else
    f := n * fact(n-1);
    return f;
end;

begin
 f:= fact(3);
end.
```

Figure 3.8 illustrates the stack states during the calling sequence of factorial(3). Initiall
main function calls factorial(3), and an activation record is created (see shaded area in
3.8, left-hand side). Then the factorial function calls itself, executing factorial(2), and

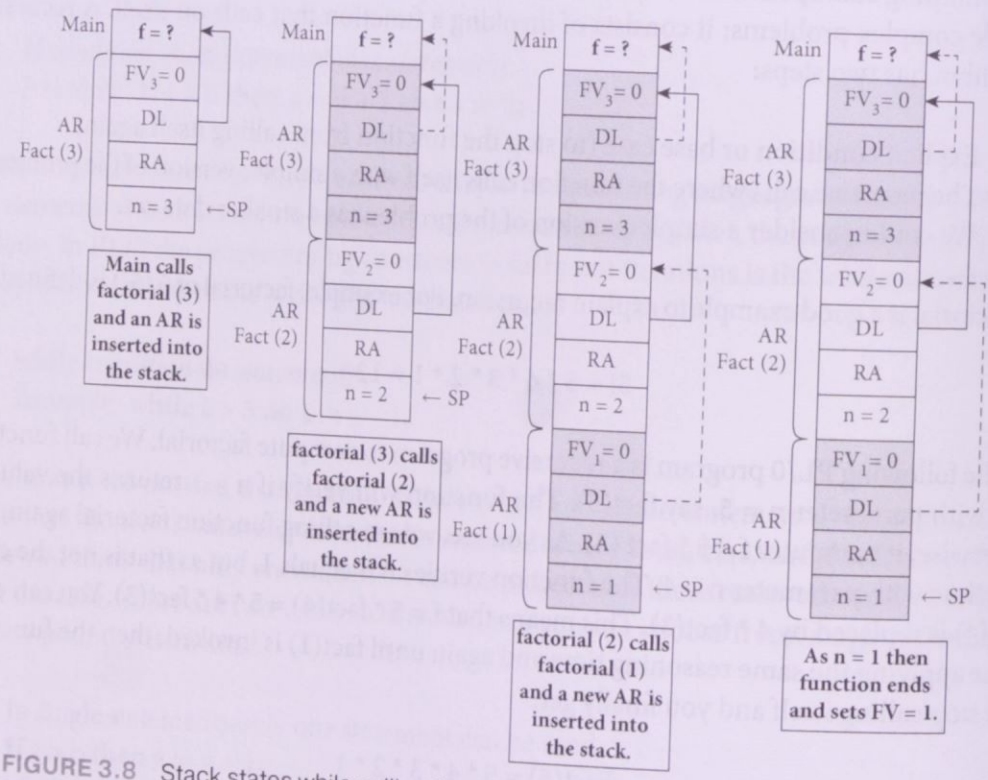


FIGURE 3.8 Stack states while calling factorial recursively.

activation record is created. Then while executing factorial(2), as the explicit condition has not been met, the factorial function calls itself again, but this time with parameter 1 (factorial(1)), and a new activation record is created. As the parameter 1 meets the explicit condition, the function stops calling itself, and the return sequence begins.

Figure 3.9 presents the states of the stack during the returning sequence or computational steps. You will observe in figure 3.9 how the returned value is passed on from one activation record to another. The value n = 1 is copied from the top of the stack into the functional value field (FV_1) and returns to factorial(2). Notice that the AR for factorial(1) is popped out, leaving $FV_1 = 1$ on top of the stack. Factorial(2) multiplies the values n^*FV_1 (2*1) found on top of the stack and stores the resulting value in the FV_2 in the AR of factorial(2). Once factorial(2) returns, its AR is popped out of the stack, leaving the value 2 stored in FV_2. Factorial(3) multiplies FV_2 times its parameter, n = 3, and the resulting value FV2 * n = 6 is stored in FV_3. When factorial(3) ends, it returns the value 6 to f.

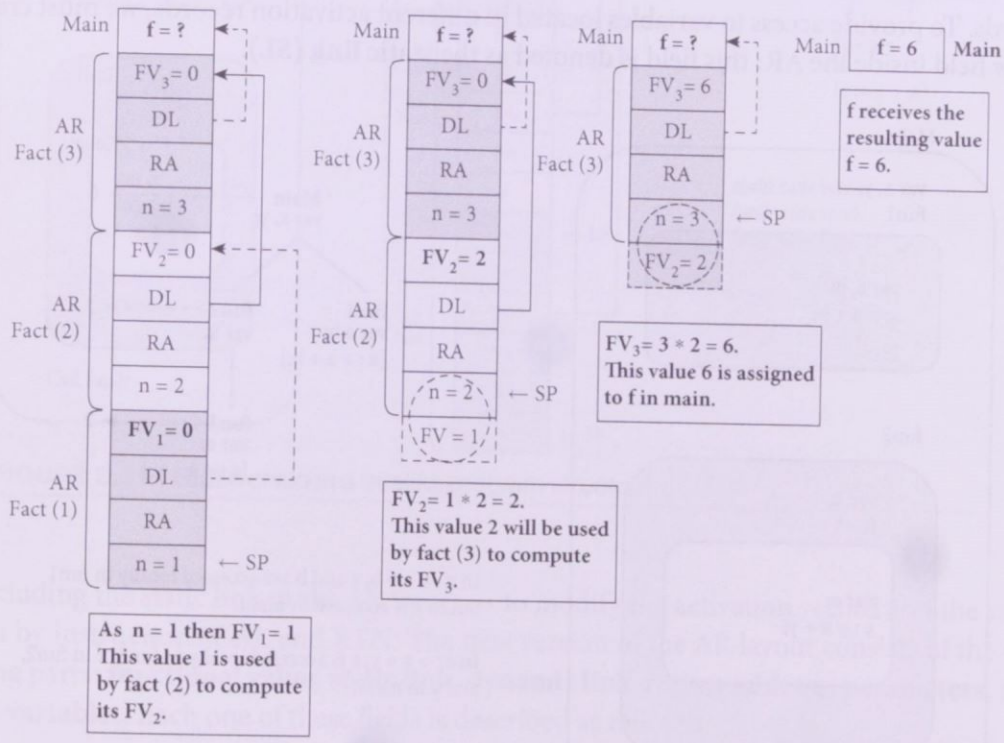


FIGURE 3.9 Stack states while factorial function is returning values.

Lexicographical Levels and Nested Programs

When programmers write programs, they use procedures to make programs look like a set of modules that can be thought of as super-instructions that can be called or used (executed) from practically anywhere within the program according to specific rules that must be followed. For instance, we cannot call a function that has not been declared. Some programming

languages, such as PL/0 and JavaScript, allow embedded functions, which means that we can declare functions in the main program, let us call them fun1 and fun2, and inside fun2 we can declare another function, fun3. This creates an embedded static structure for this program, as illustrated in figure 3.10. This treelike structure is organized in levels, and these levels are called **lexicographical levels**. In this example, main is at level 0, functions fun1 and fun2 are at level 1, and fun3 is at level 2. Exploring figure 3.10 a little further, you can observe that in each function there are variables declared, and you will see also some statements. In the lexicographical tree, underneath the function name, you will see the variables declared in each function. Also, you can see a couple of statements in the program, one within fun1 and another inside fun3. In the tree those statements are placed underneath the function name in square brackets. You might notice that in the statement **x := a + b** in fun1, **a** and **b** are accessed locally in fun1, and variable **x** is accessed in main. Similarly, in the statement **e := x + y**, **e** is accessed locally in fun3, **x** is accessed in fun2, and **y** is accessed in main. In the latter case, variables are found in different lexicographical levels, which implies different activation records. To provide access to variables located in different activation records, we must create a new field inside the AR; this field is denoted as the **static link (SL)**.

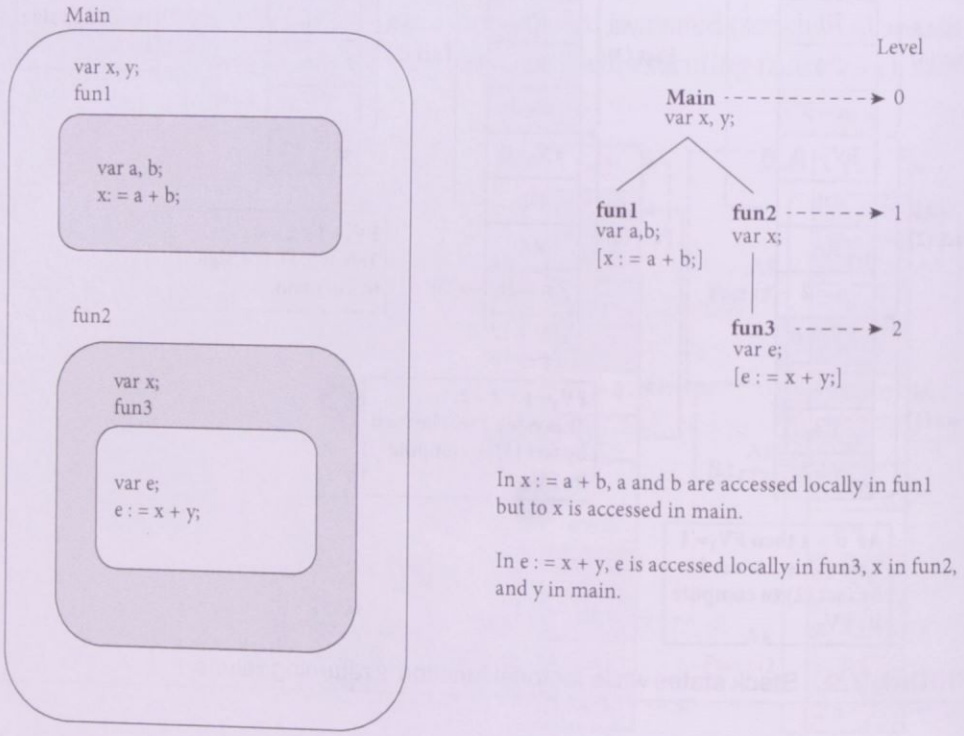


FIGURE 3.10 Treelike structure of a nested program.

In each one of the function activation records, the static link points to the base of the activation record of its ancestor or procedure/function that statically encloses the function. For our example, figure 3.11 shows that the static link of fun3 points to the base of the activation

record of fun2, and the static link in the AR of fun2 and fun1 point to the base of the activation record of main. This path is called a **static chain** and can be observed in the stack shown in figure 3.11. Also, you can see in the figure that in the stack and in the static program structure (tree), static links are shown with dashed lines. The stack in figure 3.11 is generated by the following function calls: Main calls Fun2, Fun2 calls Fun3, and Fun3 calls Fun1.

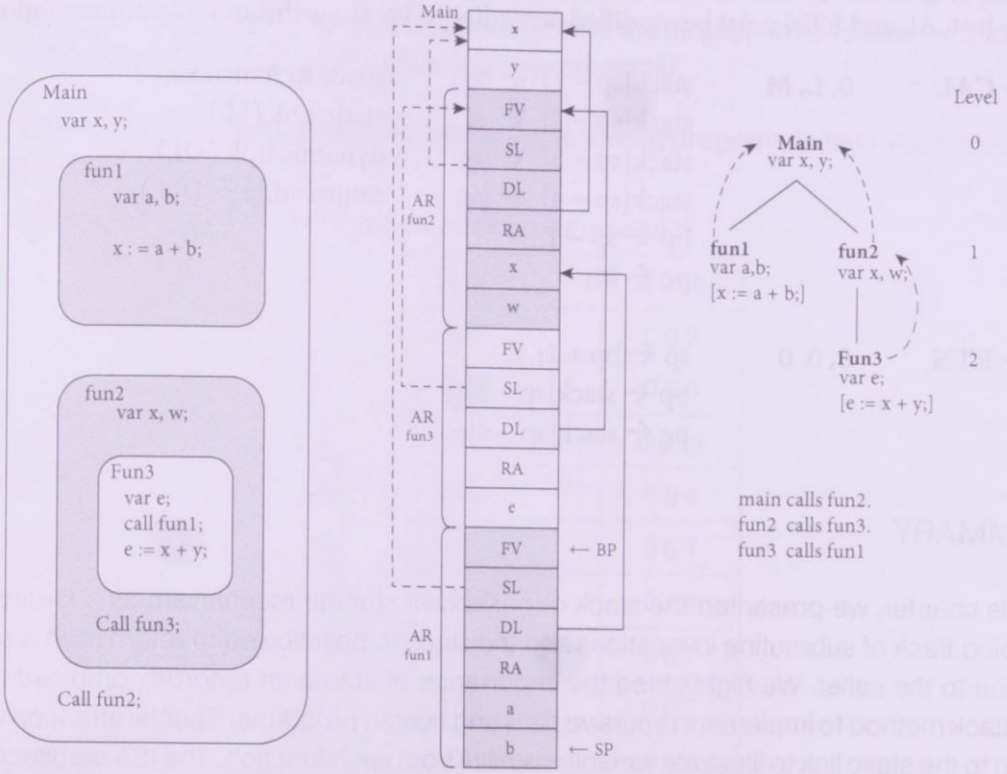


FIGURE 3.11 Static chain in a treelike program structure.

Including the static link in the AR forces us to modify the activation record and the steps taken by instructions CAL and RTN. The new version of the AR layout consists of the following parts: **functional value, static link, dynamic link**, **return address**, **parameters**, and **local variables**. Each one of these fields is described as follows:

Activation record

Functional value: Location to store the function returned value.

Static link: Points to the base of the stack frame of the procedure that statically encloses the current function or procedure (the function ancestor).

Dynamic link: Points to the base of the previous stack frame (base of the caller AR).

Return address: Points, in the code segment, to the next instruction to be execut after termination of the current function or procedure.

Parameters: Space reserved to store the actual parameters of the caller function.

Locals: Space reserved to store local variables declared in the called function (calle

As the activation record was modified with the insertion of the static link, then the steps to taken by CAL and RTN must be modified accordingly. We show this modification as follo

06 – **CAL**	**0, L, M**	stack[sp – 1] ← 0;	space to return value
		stack[sp – 2] ← sl	static link (SL)
		stack[sp – 3] ← bp;	dynamic link (DL)
		stack[sp – 4] ← pc;	return address (RA)
		bp ← sp – 1;	
		pc ← **M**;	
00 – **RTN**	**0, 0, 0**	sp ← bp + 1;	
		bp ← stack[sp – 3];	
		pc ← stack[sp – 4];	

SUMMARY

In this chapter, we presented the stack or pushdown storage mechanism as a means keeping track of subroutine invocations and the steps to be followed to return from a sub routine to the caller. We highlighted the importance of activation records combined wi the stack method to implement recursive calls and nested programs. Special attention wa given to the static link to illustrate variable visibility from each function. The ISA architectur of VM/0 was enhanced with three new instructions to implement subroutine invocation an subroutine return.

EXERCISES

1. Write a program to evaluate the expression **3 4 * 5 3 *** + using VM/0 assembly language.

2. What is the information stored in the return address and dynamic link when a function is called? And why do we need to save it in the activation record?

3. Describe the steps executed by the instructions CAL 0, 0, M and RTN 0, 0, 0 in VM/0.

4. Why do nested programs need the static link (SL) in their activation record (AR)?

5. When a function is called, do you need the instruction INC to create the activation record?

6. Using pseudocode or the programming language of your preference, write a recursive program to find the greatest common divisor of two numbers (integers). The greatest common divisor of two integers is the largest positive integer that divides both numbers without leaving any remainder.

7. Draw a picture of the stack created by the following program. Initial values are as follows:

   SP = −1, BP = 0, PC = 0. (Assume the stack grows upward.)

0	JMP	0 0 10
1	JMP	0 0 2
2	INC	0 0 6
3	LIT	0 0 13
4	STO	0 0 4
5	LIT	0 0 1
6	STO	0 1 4
7	LIT	0 0 7
8	STO	0 0 5
9	RTN	0 0 0
10	INC	0 0 6
11	LIT	0 0 3
12	STO	0 0 4
13	LIT	0 0 9
14	STO	0 0 5
15	CAL	0 0 2
16	SIO	0 0 2

Bibliographical Notes

As an alternative to the standard infix notation, in 1929 Jan Lukasiewicz[2] developed reverse Polish notation as a new method to evaluate expressions. With this new notation, the expression **(3 * 4) + (5 * 6)** would be written as **3 4 * 5 3 * +.** Reverse Polish notation could be considered the seminal idea that originated the development of recursive