UNIT II

LESSON

4

PROGRAMMING LANGUAGES

CONTENTS

4.0 Aims and Objectives

4.1 Introduction

4.2 Assembly Language

  4.2.1 Comparison of Assembly and High Level Languages

  4.2.2 Kinds of Processors

4.3 Assembler

4.4 Subroutine

4.5 Input/Output Programming

  4.5.1 Interrupt Initiated Input/Output

  4.5.2 Interrupt driven I/O Mechanism

  4.5.3 Polling

4.6 Let us Sum up

4.7 Keywords

4.8 Questions for Discussion

4.9 Suggested Readings

4.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain assembler
- Identify assembly language
- Define subroutines
- Discuss input/output programming

4.1 INTRODUCTION

Unlike the other programming languages catalogued here, assembly language is not a single language, but rather a group of languages. Each processor family (and sometimes individual processors within a

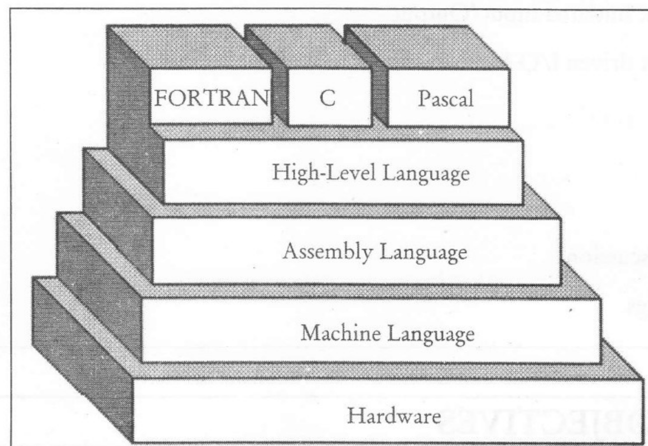
processor family) has its own assembly language. Nevertheless, assembly language is the most powerful computer programming language available, and it gives programmers the insight required to write effective code in high-level languages.

In contrast to high level languages, data structures and program structures in assembly language are created by directly implementing them on the underlying hardware. So, instead of cataloguing the data structures and program structures that can be built (in assembly language you can build any structures you so desire, including new structures nobody else has ever created), we will compare and contrast the hardware capabilities of various processor families.

4.2 ASSEMBLY LANGUAGE

A programming language that is once detached from a computer's machine language. Machine languages consist completely of numbers and are almost impossible for humans to read and write. Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers.

Every type of CPU has its own machine language and assembly language, so an assembly language program written for one type of CPU won't run on another. In the early days of programming, all programs were written in assembly language. Currently, most programs are written in a high-level language such as FORTRAN or C. Programmers still use assembly language when speed is essential or when they need to execute an operation that isn't possible in a high-level language.



The oldest non-machine language, permit for a more human readable method of writing programs than writing in binary bit patterns (or even hexadecimal patterns). Assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations implemented directly on the physical CPU. Assembly language lacks high-level amenities such as variables and functions, and it is not portable between various families of processors. Although the numbers of the above program make perfect sense to a computer, they are about as clear as mud to a human. Who would have guessed that they put a dollar sign on the screen? Clearly, entering numbers by hand is a lousy way to write a program.

Assembler languages engage a unique place in the computing world. Since most assembler-language statements are symbolic of individual machine-language instructions, the assembler-language programmer has the full power of the computer at his disposal in a way that users of other languages

do not. Because of the direct relationship between assembler language and machine language, assembler language is used when high competence of programs is needed, and especially in areas of application that are so new and amorphous that existing program-oriented languages are ill-suited for describing the procedures to be followed.

Availability

Assemblers are obtainable for just about every processor ever made. Native assemblers produce object code on the same hardware that the object code will run on. Cross assemblers produce object code on different hardware that the object code will run on.

Structure

Format: free form or column (depends on the assembly language)

Nature: procedural language with one to one correspondence among language mnemonics and executable machine instructions.

4.2.1 Comparison of Assembly and High Level Languages

Assembly languages are close to a one to one correspondence between symbolic instructions and executable machine codes. Assembly languages also comprise directives to the assembler, directives to the linker, directives for organizing data space, and macros. Macros can be used to unite several assembly language instructions into a high level language-like construct (as well as other purposes). There are cases where a symbolic instruction is translated into more than one machine instruction. But in general, symbolic assembly language instructions correspond to individual executable machine instructions.

High level languages are abstract. Typically a single high level instruction is interpreted into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions. Some early high level languages had a close correspondence between high level instructions and machine language instructions. For example, most of the early COBOL instructions translated into a very obvious and small set of machine instructions. The trend over time has been for high level languages to increase in abstraction. Modern object oriented programming languages are highly abstract (although, interestingly, some key object oriented programming constructs do translate into a very compact set of machine instructions).

Assembly language is much harder to program than high level languages. The programmer must pay consideration to far more detail and must have an intimate knowledge of the processor in use. But high quality hand crafted assembly language programs can run much faster and use much less memory and other resources than a similar program written in a high level language. Speed increases of two to 20 times faster are fairly common, and increases of hundreds of times faster are rarely possible. Assembly language programming also gives direct access to key machine features essential for implementing certain kinds of low level routines, such as an operating system kernel or microkernel, device drivers, and machine control.

High level programming languages are much easier for less skilled programmers to work in and for semi-technical managers to manage. And high level languages allow faster development times than work in assembly language, even with highly skilled programmers. Development time boost of 10 to 100 times faster are fairly common. Programs written in high level languages (especially object oriented programming languages) are much easier and less expensive to maintain than similar

programs written in assembly language (and for a successful software project, the vast majority of the work and expense is in maintenance, not initial development).

4.2.2 Kinds of Processors

Processors can broadly be divided into the categories of: CISC, RISC, hybrid, and special purpose.

Complex Instruction Set Computers (CISC) has a large instruction set, with hardware support for a wide selection of operations. In scientific, engineering, and mathematical operations with hand coded assembly language (and some business applications with hand coded assembly language), CISC processors typically perform the most work in the shortest time.

Reduced Instruction Set Computers (RISC) has a small, dense instruction set. In most business applications and in programs created by compilers from high level language source, RISC processors usually perform the most work in the shortest time.

Hybrid processors are some mixture of CISC and RISC approaches, attempting to balance the advantages of each approach.

Special purpose processors are optimized to execute specific functions. Digital signal processors, graphics chips, and various kinds of co-processors are the most common kinds of special purpose processors.

4.3 ASSEMBLER

An assembler is a program that obtains basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. A number of people call these instructions assembler language and others use the term assembly language.

Here's How it Works

- Most computers come with a particular set of very basic instructions that correspond to the basic machine operations that the computer can perform.
- The programmer can write a program using a sequence of these assembler instructions.
- This sequence of assembler instructions, recognized as the source code or source program, is then specified to the assembler program when that program is started.
- The assembler program takes each program statement in the source program and creates a corresponding bit stream or pattern (a series of 0's and 1's of a given length).
- The output of the assembler program is called the object code or object program relative to the input source program. The sequence of 0's and 1's that comprise the object program is sometimes called machine code.
- The object program can then be run (or executed) every time desired.

In the earliest computers, programmers actually wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming. Nowadays, assembler programming is used only where very efficient control over processor operations is needed. It entails knowledge of a particular computer's instruction set, however. Historically, most programs have been written in "higher-level" languages such as COBOL, FORTRAN, PL/I, and C. These languages are

easier to learn and quicker to write programs with than assembler language. The program that processes the source code written in these languages is called a compiler. Like the assembler, a compiler takes higher-level language statements and reduces them to machine code.

A newer idea in program training and portability is the concept of a virtual machine. For example, using the Java programming language, language statements are compiled into a generic form of machine language known as bytecode that can be run by a virtual machine, a kind of theoretical machine that approximates most computer operations. The bytecode can then be sent to any computer platform that has previously downloaded or built in the Java virtual machine. The virtual machine is aware of the specific instruction lengths and other particularities of the platform and ensures that the Java bytecode can run.

4.4 SUBROUTINE

A subroutine is a self-contained sequence of instructions that does away the computational tasks. A subroutine is employed a number of times during the execution of a program. Wherever a subroutine is called to perform its function, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is reverted to the main program. Various names are assigned to the instruction that transfers program control to a subroutine. For example, call subroutine, jump to subroutine, branch to subroutine do. A call subroutine instruction comprises of an operation code with an address that specifies the beginning of the subroutine. As such two operations are included for execution of instruction (1) storage of the address of next instruction available in the program counter (the return address) in a temporary location so that the subroutine knows where to return, and (2) transfer of control to the beginning of the subroutine. The last instruction of every subroutine, referred as return from subroutine, causes transfer of returns address from the temporary location into the program counter. Consequently, program control is transferred to the instruction whose address was originally stored in the temporary location.

The temporary location for storing the return address differs from one computer to another. Some store it in a fixed location in memory, some store it in the first memory location of the subroutine, some store it in a processor rights and some store it in a memory stack. However, the best way is to store the return address in a memory stack. This is advantageous when successive subroutines are called because the sequential return addresses can be pushed into the stack. The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter.

When a subroutine stack is employed, all return addresses are automatically stored by the hardware in one unit. This does away the problem of recalling where the return address was stored.

4.5 INPUT/OUTPUT PROGRAMMING

Programmed Input/Output are useful I/O method for computers where hardware costs need to be minimized. The Input or Output operation in such cases may involve:

- Transfer of data from I/O device to the CPU registers.
- Transfer of data from CPU registers to memory.

- In addition, in a programmed I/O method the responsibility of CPU is to constantly check the status of the I/O device to check whether it has become free (in case output is desired) or it has finished inputting the current series of data (in case input is going on). Thus, Programmed I/O is a very time consuming method where CPU wastes lot of time for checking and verifying the status of an I/O device. Let us now try to focus how this Inuit-Output is performed. Figure 4.1 gives the block diagram of transferring a Block of data word by word using programmed I/O technique.

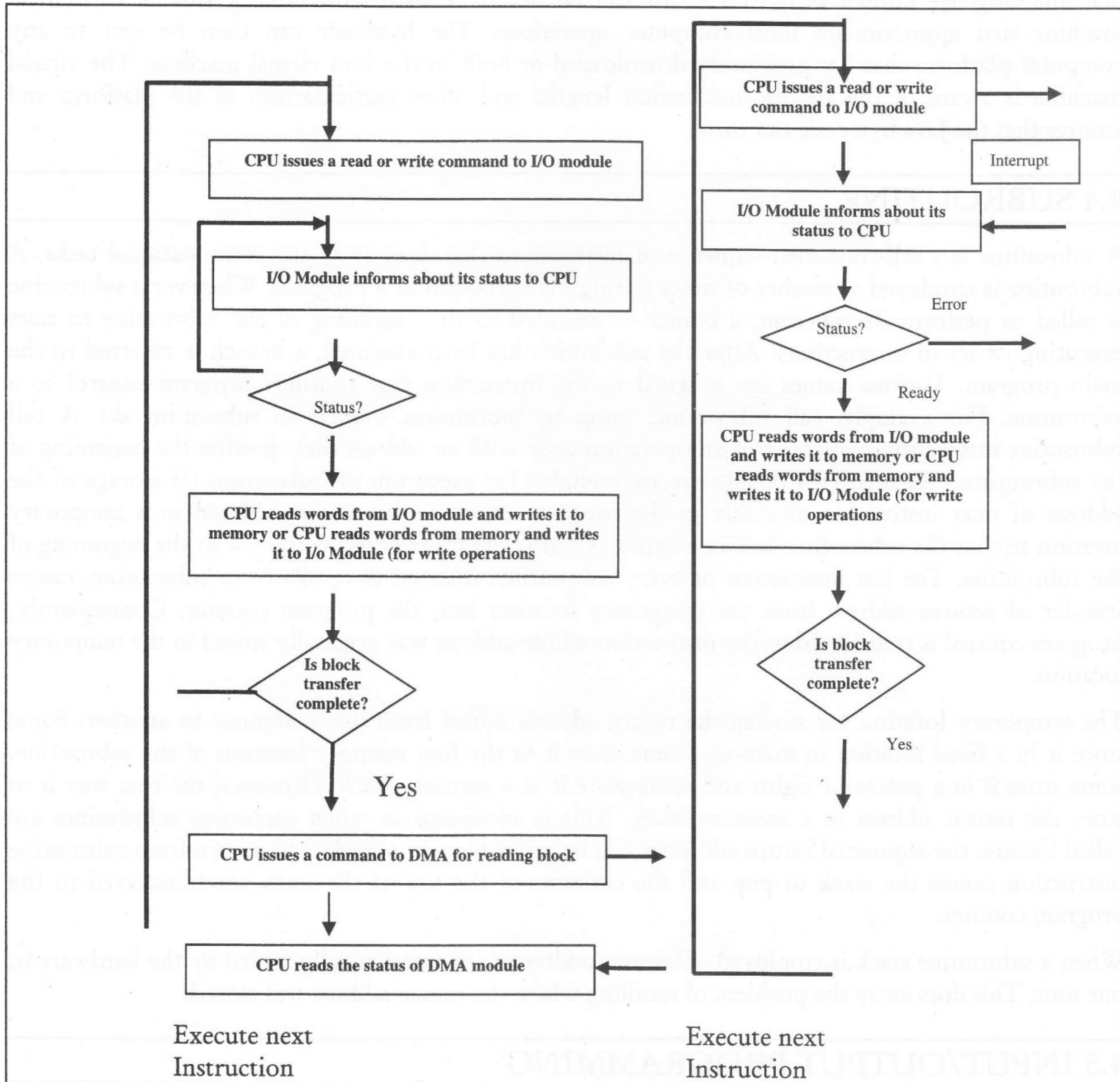


Figure 4.1: CPU Performs the Next Instruction

I/O Instruction: To carry out Input/Output CPU issues I/O related instructions. These instructions consist of two components:

- The address of the Input/Output device specifying the I/O device and I/O module; and
- An Input/Output command.

There are four types of I/O commands which can be classified as:

(a) Control (b) Test (c) Read (d) Write

Control commands are device specific and are used to control the specific instructions to the device. E.g. a magnetic tape requires rewinding or moving forward by a block. Test commands checks the status such as if a device is ready or not or is in error condition. The read command is used for input of data from input device and write command is used for output of data to output device.

The other part of I/O instruction is the address of the I/O device, In systems with programmed I/O the I/O module, the main memory and the CPU normally share the system bus. Thus each I/O module should interpret the address lines to determine if the command is for itself. Or in other wordsHow does CPU specifies which device to access? There are two methods of doing so. These are called memory mapped I/O and I/O- mapped I/O.

If we use the single address space for memory locations and I/O devices, i.e. the CPU treats the status and data registers of I/O module as memory locations. Then memory and I/O devices can be accessed using the same instructions. This is referred to as memory mapped I/O. For a memory mapped I/O only a sinlge READ and a single WRITE line are needed for memory or I/O module read or write operations. These lines are activated by CPU for either memory access or I/O device access. Figure 4.2 shows the memory mapped I/O system structure. This scheme is used in Motorola 68000.

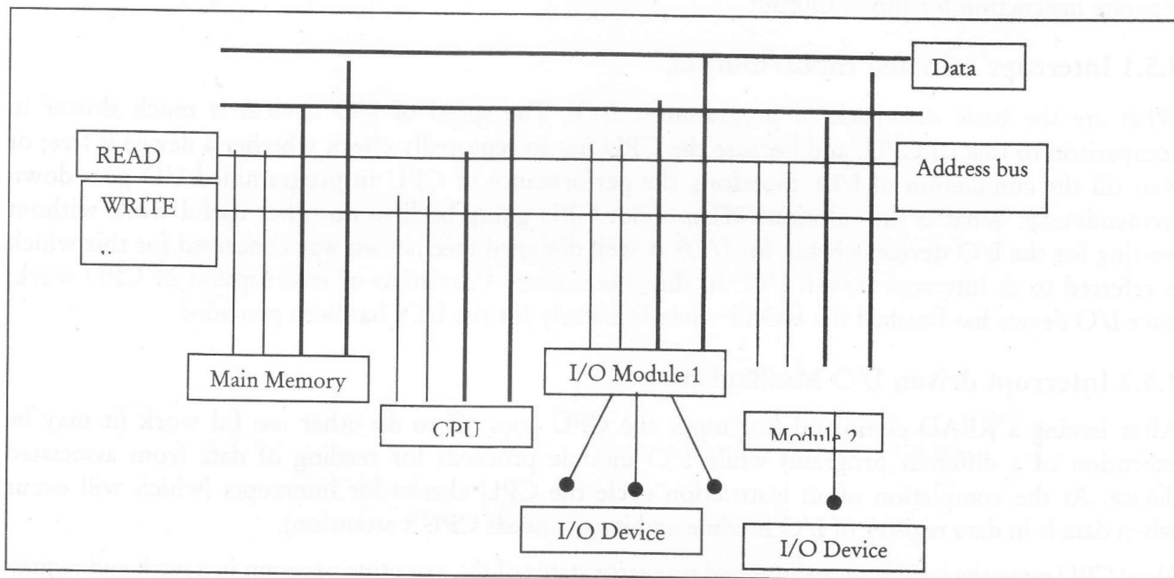


Figure 4.2: Structure of Memory Mapped I/O

In I/O-mapped I/O the I/O devices and memory are addressed separately (Refer Figure 4.3). There are separate control lines for memory and I/O device read or write operations, thus, a memory reference instruction do not affect an I/O device. Here separate Input/Output Instructions are needed which cause data transfer between addressed I/O module and CPU. This structure is used in Intel 8085 and 8086 series.

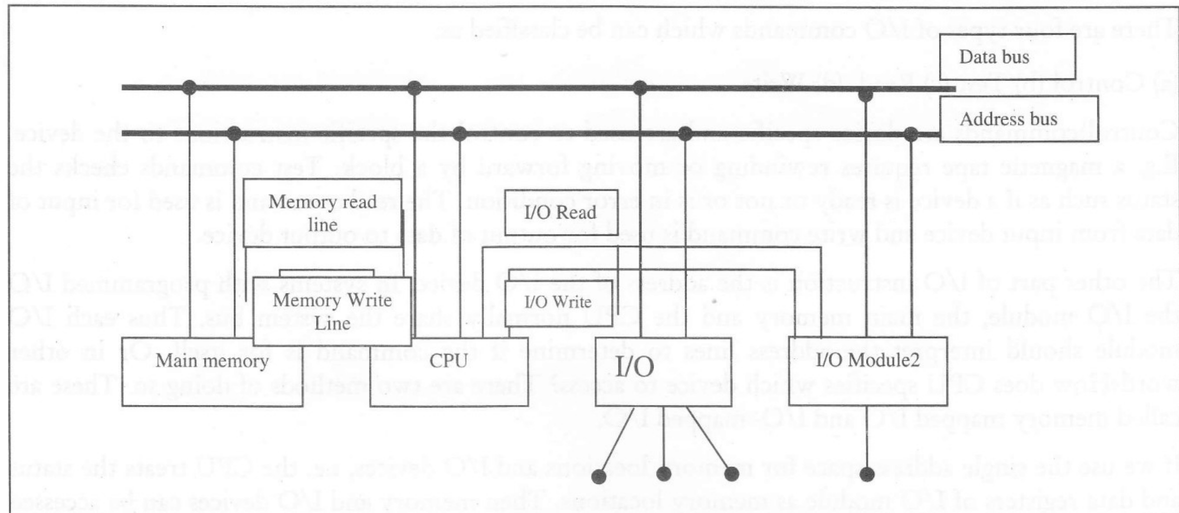


Figure 4.3: Structure of I/O Mapped I/O or Isolated I/O Scheme

Please note the different of requirements as in the case of memory mapped I/O the READ instructions may bring data to or from memory or I/O module, while in I/O-mapped I/O we need to have separate instruction for Input/Output.

4.5.1 Interrupt Initiated Input/Output

What are the basic drawback of programmed I/O? The speed of I/O devices is much slower in comparison to that of CPU, and because the CPU has to repeatedly check whether a device is free; or wait till the completion of I/O, therefore, the performance of CPU in programmed I/O goes down tremendously. What is the solution? What about CPU going back to do other useful work without waiting for the I/O device is ready for I/O? A well designed mechanism was conceived for this which is referred to as Interrupt driven I/O. In this mechanism, Provisions of interruption of CPU work, once I/O device has finished the I/O or when it is ready for the I/O, has been provided.

4.5.2 Interrupt driven I/O Mechanism

After issuing a READ command (for input the CPU goes off to do other use ful work (it may be execution of a different program) while I/O module proceeds for reading of data from associated device. At the completion of an instruction cycle the CPU checks for interrupts (which will occur when data is in data register of I/O module and it now needs CPU's attention).

Now CPU saves the important register and processor status of the executing program in a stack and request I/O device to provide its data which is placed on data bus by I/O device, After taking t he required action with the data, the CPU can go back to the program it was executing before the interrupt.

Interrupt: The term interrupt loosely is used for any execeptional event that causes temporary transfer of control of CPU from one program to the other which is causing the interrupt. Interrupts are primarily issued on:

- Initiation of Input/Output operation
- Completion of an Input/Output operation
- Occurrence of hardware of software errors.

Interrupts can be generated by various sources internal or ext external to the CPU. An interrupt generated internally by CPU is sometimes termed as Traps. The traps are normally result of programming errors such as division by zero while execution of a program.

The two key issues in Interrupt driven Input/Output are:

- To determine the device which has issued an interrupt
- In case of occurrence of multiple interrupts which one to be processed first.

There are several solution to these problems. The simplest of them is to provide multiple interrupt lines which will result in immediate recognition of the interrupting device. The priorities can be assigned to various interrupts and the interrupt with highest priority should be selected for service in case multiple interrupt occurs. But providing multiple interrupt lines is an impractical approach because only a few lines of the system bus can be devoted for the interrupt. Other methods for this are software poll, daisy chaining and bus arbitration.

4.5.3 Polling

In this scheme on occurrence of an interrupt, CPU starts executing a software routine termed as interrupt service program or routine which poll to each I/O module to determine which I/O module has caused the interrupt. This may be achieved by reading the status register of the I/O modules. The priority here can be implemented easily by defining the spooling sequence, since the device polled first will have higher priority. Please note that after identifying the device the next set of instructions to be executed will be the device service routines of the device, resulting in the desired input or output.

As far as daisy chaining is concerned, we have one Interrupt Acknowledge line, which is chained through various interrupt devices. There is just one Interrupt Request line. On receiving an Interrupt Request the Interrupt Acknowledge line is activated which in turn passes this signal device by device. The first device which has made the interrupt request thus graphs thus graphs the signal and responds by putting a word which is normally an address of interrupt servicing program of a unique identifier on the data lines. This word is also referred to as interrupt vector. This address or identifier in turn is used for selecting an appropriate interrupt-servicing program. The daisy chaining has an in-built priority scheme which is determined by the sequence of devices on interrupt acknowledge line.

In bus arbitration technique, the I/O module first first need to control the bus and only after that can request for an interrupt. In this scheme, since only one of the modules can control the bus therefore, only one request can be made at a time. The interrupt request is acknowledged by the CPU on response of which I/O module places the interrupt vector on the data lines.

Check Your Progress

State whether the following statements are true or false:

1. Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers.
2. Assemblers are not available for just about every processor ever made.
3. Macros can be used to combine several assembly language instructions into a high level language-like construct.
4. High level programming languages are not easier for less skilled programmers to work in and for semi-technical managers to supervise.
5. A subroutine is a self-contained sequence of instructions that does away the computational tasks.

4.6 LET US SUM UP

Assembler languages occupy a unique place in the computing world. Since most assembler-language statements are symbolic of individual machine-language instructions, the assembler-language programmer has the full power of the computer at his disposal in a way that users of other languages do not. High level languages are abstract. Typically a single high level instruction is translated into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions. Some early high level languages had a close correspondence between high level instructions and machine language instructions. An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term assembly language. A subroutine is a self-contained sequence of instructions that does away the computational tasks. A subroutine is employed a number of times during the execution of a program. Wherever a subroutine is called to perform its function, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is reverted to the main program. Programmed Input/output are useful I/O method for computers where hardware costs need to be minimized.

4.7 KEYWORDS

Complex Instruction Set Computers (CISC): It has a large instruction set, with hardware support for a wide variety of operations.

Reduced Instruction Set Computers (RISC): It has a small, compact instruction set.

Hybrid: These processors are some combination of CISC and RISC approaches, attempting to balance the advantages of each approach.

Special Purpose: These processors are optimized to perform specific functions.

4.8 QUESTIONS FOR DISCUSSION

1. What is Assembly language?
2. What is the difference between Assembly Language and High Level Language?
3. Explain subroutines.
4. Explain interrupt initiated input/output.

Check Your Progress: Model Answers

1. True
2. False
3. True
4. False
6. True

4.9 SUGGESTED READINGS

Sajjan G. Shiva; *Computer Design and Architecture*; Marcel Dekker

Silvia Melitta Mueller, Wolfgang J. Paul; *Computer Architecture*; Springer

Joseph D. Dumas II; *Computer Architecture*; CRC Press

Nicholas P. Carter; *Schaum's Outline of Computer Architecture*; Mc. Graw-Hill Professional

LESSON

5

REGISTER TRANSFER LANGUAGE

CONTENTS

5.0 Aims and Objectives

5.1 Introduction

5.2 Types of Register

5.3 Register Transfer

5.4 Data Transfer and Manipulation

- 5.4.1 Data Transfer Instructions
- 5.4.2 Data Manipulation Instructions
- 5.4.3 Program Control

5.5 Conditional Branch Instructions

5.6 Program Interrupt

5.7 Microprocessor Organisation of 8086

5.8 Let us Sum up

5.9 Keywords

5.10 Questions for Discussion

5.11 Suggested Readings

5.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Describe register transfer
- Explain data transfer and manipulation
- Discuss program control
- Explain microprocessor organization of 8086

5.1 INTRODUCTION

A register is a group of Flip Flops, i.e., a register is also a storing device but its capability of storing data is more than that of a single Flip Flop. A register can store as any bits as the number of Flip Flops

it contains. So an 'n' bit register must be capable of storing 'n' number of bits, i.e., it has 'n' number of Flip Flops. It can have logic too so it has processing capability too.

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital system design invariably uses a modular approach. The modules are constructed from such digital components such as registers, decoders, arithmetic elements and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called Microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

The term "register transfer" implies the availability and hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. A register transfer language is a system for expressing in symbolic form the microoperation tool for describing the internal organization and digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

5.2 TYPES OF REGISTER

- ***Shift Registers:*** A register capable of shifting its binary information either to the right or to the left is called a shift register.
- ***Serial Shift Register:*** A serial register is one that shifts the information stored in it, one by one, i.e., one bit after another. The shift register in simplest form can be shown as follows (i.e., containing only Flip Flops, in cascade and number of logic gates.)

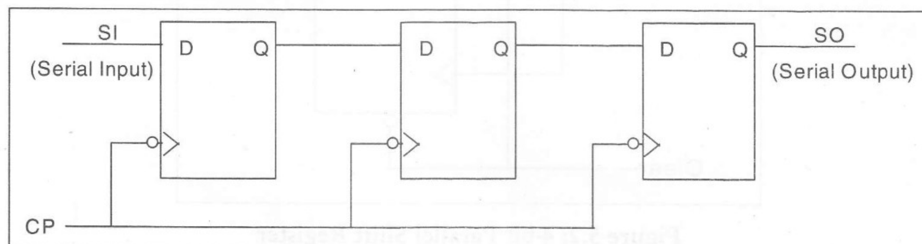


Figure 5.1: Serial Shift Register

The bubble before the CP input shows that triggering of Flip Flops occurs at falling edge of CP.

The above type of Shift Register provides serial transfer.

- ***Parallel Shift Register:*** A parallel shift register shifts the bits in a parallel mode. In parallel mode, information present at the input can be transferred to the output, simultaneously, during clock period.

  Simplest example of 4-bit Shift Register is:

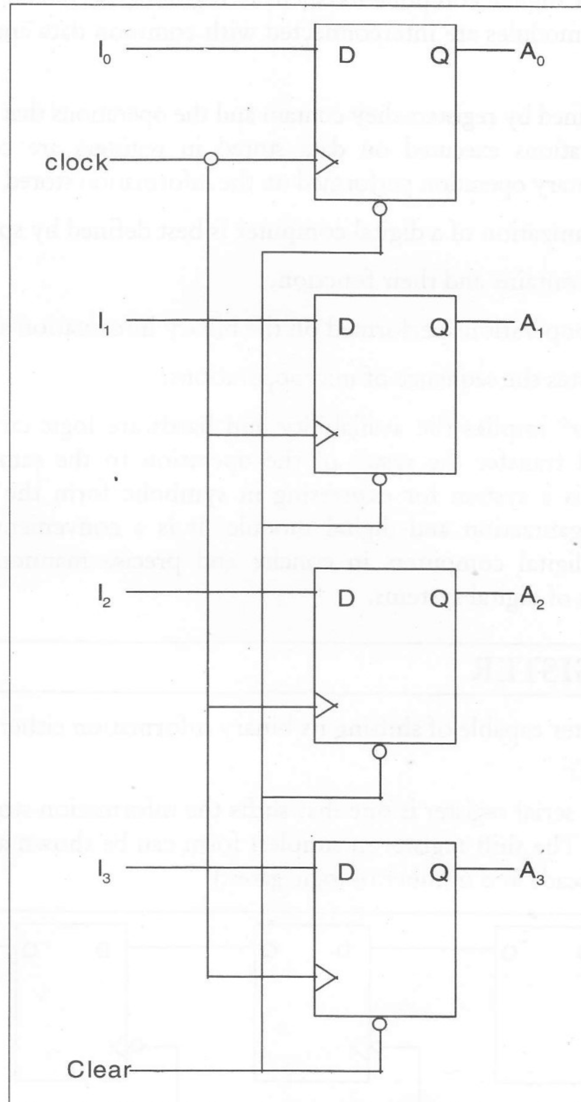


Figure 5.2: 4-bit Parallel Shift Register

So it is clear that in serial transfer the information passes one bit at a time and in parallel transfer, all the bits present at the input appear at the output on the application of clock pulses.

- ***Bidirectional Shift Register:*** A shift register that can shift the information in both ways, i.e., from right to left and from left to right, by providing a control signal. The circuit diagram and working of the circuit is shown below:

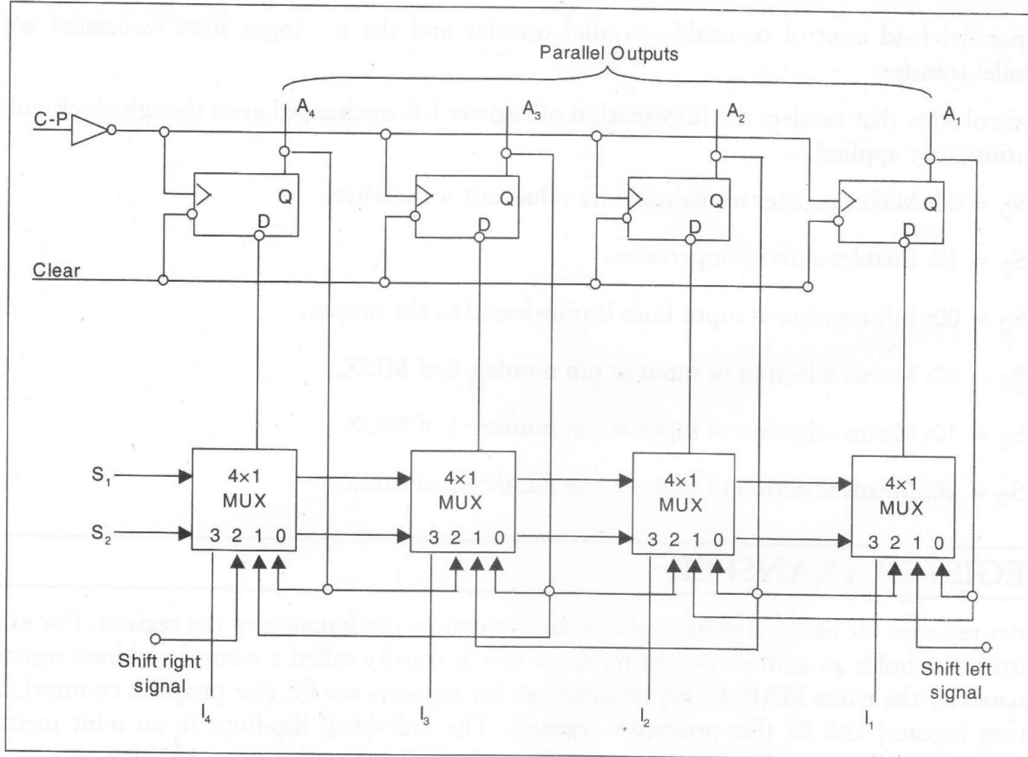


Figure 5.3: Bidirectional Parallel Shift Register

Table 5.1: Truth Table of the Adjoining Figure

Control Signals		Register Operation
S_1	S_2	
0	0	No Change
0	1	Shift left
1	0	Shift right
1	1	Parallel load

CP: Clock Pulse Signal

Clear: This signal clears all the Flip Flops when it is '0' (Note the "bubble indication")

The diagram shows Bidirectional Shift Register with parallel load. The descriptions are as follows:

1. A clear control to clear the register to 0.
2. A CP input for clock pulses to synchronize all operations.
3. A shift right control to enable the shift right operation.
4. A shift left control to enable shift left operation.

5. A parallel load control to enable parallel transfer and the n - input lines associated with the parallel transfer.
6. Control state that renders the information of register left unchanged even though clock pulses are continuously applied.

   S_1S_0 = 00: Makes register to maintain its value as it was earlier.

   S_1S_0 = 10: Enables shift left operation.

   S_1S_0 = 00: Information at input lines is transferred to the output.

   S_1S_0 = 00: Means selection of input at pin number 0 of MUX.

   S_1S_0 = 10: Means selection of input at pin number 1 of MUX.

   S_1S_0 = 00: Means selection of input at pin number 2 of MUX.

5.3 REGISTER TRANSFER

Computer registers are designated by capital letters to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR. Other designations for registers are PC (for program counter), IR (for instruction register) and RI (for processor register). The individual flip-flops in an n-bit register are numbered in sequence from 0 through n – 1, starting from 0 in the rightmost position and increasing the numbers toward the left.

A 16-bit register is partitioned into two parts. Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC(L) refers to the low order byte and PC(H) to the high order byte. The statement $R_2 \leftarrow R_1$ denotes a transfer of the content of register R_1 into register R_2. The content of the source register R_1 does not change after the transfer. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an "if then" statement.

$$\text{If } P=1 \text{ then } R_2 \leftarrow R_1$$

where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$$P: R_2 \leftarrow R_1$$

Bus System

A typical digital computer has many registers and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is

transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. In general, a bus system will multiplex registers of a bit each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be $K \times 1$ since it multiplexes k data lines. A bus system can be constructed with 'three-state gates' instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. The one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three state buffer gate is shown in Figure 5.4. The control input determines the output. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and it goes to high-impedance state, regardless of the value in the normal input.

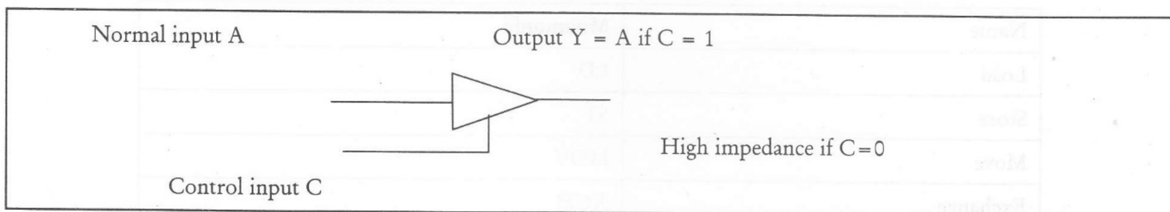


Figure 5.4: Graphic Symbols for Three-state Buffer

5.4 DATA TRANSFER AND MANIPULATION

Computers are a boon to mankind when its come to carrying out various computational tasks, for they provide an extensive set of instructions to give the user the flexibility to perform these tasks. The basic difference between the instruction set of various computers is the way the operands are determined from the address and mode fields. But there are certain basic operations that are included in the instructions set every computer. Such a basic set of operation available in a typical computer can be classified in the three categories:

1. Data transfer instructions.
2. Data manipulation instructions.
3. Program control instructions.

5.4.1 Data Transfer Instructions

As the name suggests data transfer instructions are meant for transfer for data from one location to another, keeping the binary information intact. The useful transfer are between memory and processing registers, between processor registers and input or output, and between the processor registers themselves. Each instruction is accompanied with the mnemonic symbol which is different in different computers for the same instruction name. Table 5.2 gives a list of eight data transfer instructions used in many computers.

The "load" instruction represents a transfer from memory to a processor register, usually an "accumulator" where as the store instruction designates a transfer from a processor register into memory. The move instruction is employed in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. Swapping of information between to registers of a register and memory word its accomplished by using the exchange instruction. The input and output instructions cause transfer of data among processor registers and input or output terminals. The push and pop instructions take care of transfer of data between processor registers and a memory stack.

To distinguish with between the various address modes, the mnemonic symbol are modified by assembly language conventions. For instance, the mnemonic for load immediate becomes LDI. Other assembly language conventions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. What ever may be the case, the important thing is to realize that each instruction can occur with a variety of addressing modes.

Table 5.2: Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
POP	POP

It is imperative to be familiar with the addressing mode used, as well as the type of instructions available in the computer so as to writer assembly language programs for the computer.

5.4.2 Data Manipulation Instructions

Data manipulation instructions are those that perform operations on data and are a help in computation done on computers provide the computational capabilities for the computer.

1. Arithmetic instructions.
2. Logical and bit manipulation instructions.
3. Shift instructions.

Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most of the computers carry instructions for all four operations. For computers which have only addition and possibly subtraction instructions, the other two operations i.e. multiplication and division must be generated by means of software subroutines. These four basic arithmetic operations are adequate for providing solutions to scientific problems when expressed in terms of numerical analysis methods.

Table 5.3 shows a list of typical arithmetic instructions. The increment instruction adds 1 to the value stored in a register or memory word. A unique feature of increment operation when executed in processor register is that a binary number all 1's on incrementing produces a result of all 0's. Similarly in case of decrement instruction a number of all 0's, when decremented produces a number with all 1's.

The four kind of instruction may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in The definition of the operation code include the data type that is in processor registers during the execution these arithmetic operations. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

Table 5.3: Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed-2's complement form.

Logical and Bit Manipulation Instructions

Logical instructions are useful for performing binary operations on strings of bits stored in registers. These instructions consider each bit of the operand individually and treat it as a Boolean variable. Their proper application facilitates changing in bit values, clearing or inserting new bit values into operands stored in registers or memory words.

Some typical logical and bit manipulation instructions are listed in Table 5.4. The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the l's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on each bit of the operands separately. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented. The three logical instructions are usually applied to do just that.

Table 5.4: Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions. Another example that can be cited is of a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

Shift Instructions

Shift instructions are used to shift the content of an operand from left to right our vice-a-versa. The bit shifted in at the end of the word determines the type of shift used. Any of the logical shifts, arithmetic shifts, or rotate-type operations can be specified by shift instruction. In any case the shift may be to the right or to the left.

Four types of shift instructions are listed below in Table 5.5. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts are used in conformity with the rules for signed-2's complement numbers. As a rule the arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unaltered. This is a shift-right operation wherein the end bit remains unchanged the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction. For this reason many computers do not provide a distinct arithmetic shift-left instruction when the logical shift-left instruction is already available.

A circular shift is produced by the rotate instructions. Unlike as a logical shift where bit shifted out at one end of the word are lost, here bits are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

Table 5.5: Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL

Contd...

Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

5.4.3 Program Control

Memory locations are storage house for instructions. When processed in the CPU, the instructions are fetched from consecutive memory locations and implemented. Each time an instruction is fetched from memory, the program counter is simultaneously incremented with the address of the next instruction in sequence. Once a data transfer or data manipulation instruction is executed, control returns to the fetch cycle with he program counter containing the address of the instruction next in sequence. In case of a program control type of instruction execution of instruction may change the address value in the program counter and cause the flow of control to be altered.

The conditions for altering the content of the program counter, are specified by program control instruction, and the conditions for data-processing operations are specify by data transfer and manipulation instructions. As a result of execution of a program control instruction, a change in value of program counter occurs this causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments. Some typical program control instructions are listed in Table 5.6. The branch and jump instructions are identical in their use but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction, as a name denotes, causes a branch to the specified address without any conditions. On the contrary the conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter remains unaltered and the next instruction is taken from the next location in sequence.

Table 5.6: Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

The skip instruction does not require an address field and is, therefore, a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. This is achieved by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus, a skip-branch pair of instructions auses a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

The call and return instructions are used in conjunction with subroutines. The compare instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. In a similar fashion the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition.

The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end carry C8 is 1 .It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

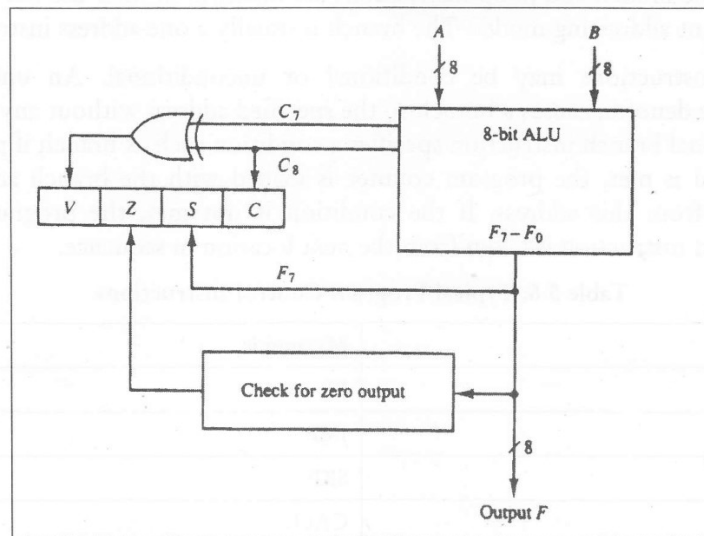


Figure 5.5: Status Register Bits

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A = B. This is so because x x = 0, and the exclusive-OR of two equal operands gives an all-0's result which sets the Z bit.

A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let A = 101x1100, where x is the bit to be checked. The AND operation of A with B = 00010000 produces a result 000x0000. If x = 0, the Z status bit is set, but if x = I, the Z bit is cleared since the result is not zero.

5.5 CONDITIONAL BRANCH INSTRUCTIONS

The commonly used branch instructions are listed below in Table 5.7. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is met, the address specified by the instruction receives program control. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions. The zero status bit is employed for testing if the result of an ALU operation is equal to zero or not. The carry bit is employed to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the most significant bit of the output from the ALU. S = 0 denotes a positive sign and S = 1, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It is worth noticeable that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's complement representation.

Table 5.7: Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
Unsigned compare conditions ($A - B$)		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

It has been discussed earlier that the compare instruction performs a subtraction of two operands, say A - B. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides Information about the relative magnitude of A and B. Some computers provide conditional branch instructions that can be applied right after the execution of a

compare instruction. The specific conditions to be tested depend on whether the two numbers A and B are considered to be unsigned or signed numbers.

The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and -128. The subtraction of two numbers is the same whether they are unsigned or in signed-2's complement representation. Let A = 11110000 and B = 00010100. To perform A -B, the ALU takes the 2's complement of B and adds it to A.

The compare instruction updates the status bits as shown. C = 1 because there is a carry out of the last stage. S = I because the leftmost bit is 1. V = 0 because the last two carries are both equal to 1, and Z = 0 because the result is not equal to 0.

If we assume unsigned numbers, the decimal equivalent of A is 240 and that of B is 20. The subtraction in decimal is 240-20 = 220. The binary result 11011100 is indeed the equivalent of decimal 220. Since 240 > 20, we have that A > B and A f B. These two relations can also be derived from the fact that status bit C is equal to I and bit Z is equal to 0. The instructions that will cause a branch after this comparison are BHI (branch if higher), BHE (branch if higher or equal), and BNE (branch if not equal).

If we assume signed numbers, the decimal equivalent of A is-16. This is because the sign of A is negative and 11110000 is the 2's complement of 00010000, which is the decimal equivalent of +16. The decimal equivalent of B is +20. The subtraction in decimal is (-16) - (+20) = -36. The binary result 11011100 (the 2's complement of 00100100) is indeed the equivalent of decimal -36. Since (-16)< (+20) we have that A< B and A [1] B. These two relations can also be derived from the fact that status bits S = 1 (negative), V = 0 (no overflow), and Z = 0 (not zero). The instructions that will cause a branch after this comparison are BLT (branch if less than), BLE (branch if less or equal), and BNE (branch if not equal).

It should be noted that the instruction BNE and BNZ (branch if not zero) are identical. Similarly, the two instructions BE (branch if equal) and BZ (branch if zero) are also identical.

5.6 PROGRAM INTERRUPT

Program interrupt can be described as a transfer of program control from a currently running program to another service program on a request generated externally or internally. After the service program is executed, the control returns to the original program.

The interrupt procedure is identical to a subroutine call except for three variations: (1) The interrupt is usually generated by an internal or external signal rather than from the execution of an instruction (except for software interrupt); (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and (3) an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

It is imperative for the CPU to return to the same state that it was when interrupt occurred after the program interrupted and the service routine has been executed. The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter.
2. The content of all processor registers.
3. The content of certain status conditions.

The collection of all status bit conditions in the CPU is referred as a program status word or PSW. The PSW is stored in a separate hardware register and contains the status information that

characterizes the state of the CPU. It is inclusive of the status bits from the last ALU operation and specifies the interrupts that are allowed to occur and whether the CPU is operating in a supervisor or user mode. Most of the computers have a resident operating system that controls and supervises all other programs in the computer. When the CPU is executives a program that is part of the operating system, it is referred to be in the supervisor or system mode. Certain instructions are executed in this mode only. The CPU is normally in the user mode when user programs are executed. Special status bits in the PSW determine the mode advantage the CPU is operating at any given time.

In certain computers only the program counter is stored when responding to an interrupt. The service program must then include instructions to store status and register content before these resources are used. Very few computers store both program counter and all status and register content in response to an interrupt. Most computers store the program counter and the PSW only. Some computers may have two sets of processor registers within the computer, one for each CPU mode. In this way, when the program switches from the user to the supervisor mode (or vice versa) in response to an interrupt, storing the contents of processor registers is not required as each mode uses its own set of registers.

The hardware procedure for processing an interrupt is very similar to the execution of a subroutine call instruction.

Types of Interrupts

Interrupts can be classified into the major types as given below:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Various examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure. Time-out interrupt may result from a program that is in an endless loop and thus consumes more time its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases. Internal interrupts arise when an instruction or data is used illegally or erroneously. These interrupts are also known as traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. Occurrence of internal errors is usually a resultant of a premature termination of the instruction execution. Remedial majors to be taken are again determine by service program that processors the internal interrupts.

To distinguish between internal and external interrupts, the internal interrupt is generated by some exceptional condition caused by the program itself rather than by an external event. Internal interrupts are synchronous with the program while external interrupts are asynchronous. On rerunning of the program, the internal interrupts will occur in exactly same place each time. On the contrary external interrupts being dependent on external conditions, are independent of the program being executed at the time.

External and internal interrupts are generated from signals that occur in the hardware of the CPU. On the contrary, a software interrupt is initiated during execution of an instruction. In precise terms, software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be called to function by the programmer to initiate an interrupt procedure at any desired point in the program. Usages of software interrupt is mostly associated with a supervisor call instruction. This

instruction is meant for switching from a CPU user mode to the supervisor mode. Certain operations in the computer are privileged to be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode. When an input or output transfer is required, the request for the supervisor mode is sent by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. This information must be passed to the operating system from the calling program so as to specify the particular task requested.

5.7 MICROPROCESSOR ORGANISATION OF 8086

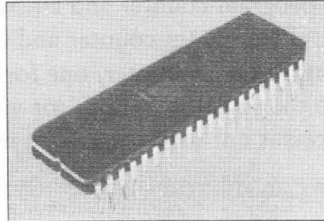


Figure 5.6: Intel Corporation's 8086 Microprocessor

- The 8086 proclaim in 1978, was the first 16-bit microprocessor introduced by Intel Corporation.
- The 8086 is internally a 16-bit MPU and externally it has a 16-bit data bus. It has the aptitude to address up to 1 Mbyte of memory via its 20-bit address bus.
- In calculation, it can address up to 64K of byte-wide input/output ports.
- It is manufactured by high-performance metal-oxide semiconductor (HMOS) technology, and the circuitry on its chip is equivalent to approximately 29,000 transistors.
- The 8086 is housed in a 40-pin dual in-line package.

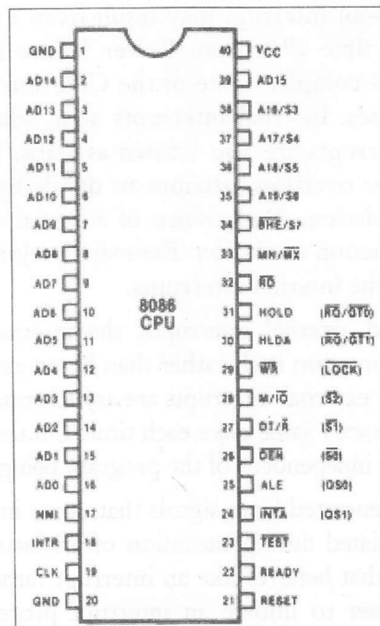


Figure 5.7: Pin layout of the 8086 Microprocessor

Minimum-Mode and Maximum-Mode

The 8086 can be configured to work in either of two modes:

- The minimum mode is chosen by applying logic 1 to the MN/MX input lead. It is typically used for smaller single microprocessor systems.
- The maximum mode is elected by applying logic 0 to the MN/MX input lead. It is typically used for larger multiple microprocessor systems.
- Depending on the mode of operation preferred, the assignments for a number of the pins on the microprocessor package are changed. The pin functions specified in parentheses pertain to the maximum-mode.
- We will only converse minimum-mode operation of the 8086. In minimum mode, the 8086 itself provides all the control signals needed to implement the memory and I/O interfaces (see Fig. 8-3). In maximum-mode, a separate chip (the 8288 Bus Controller) is used to help in sending control signals over the shared bus).

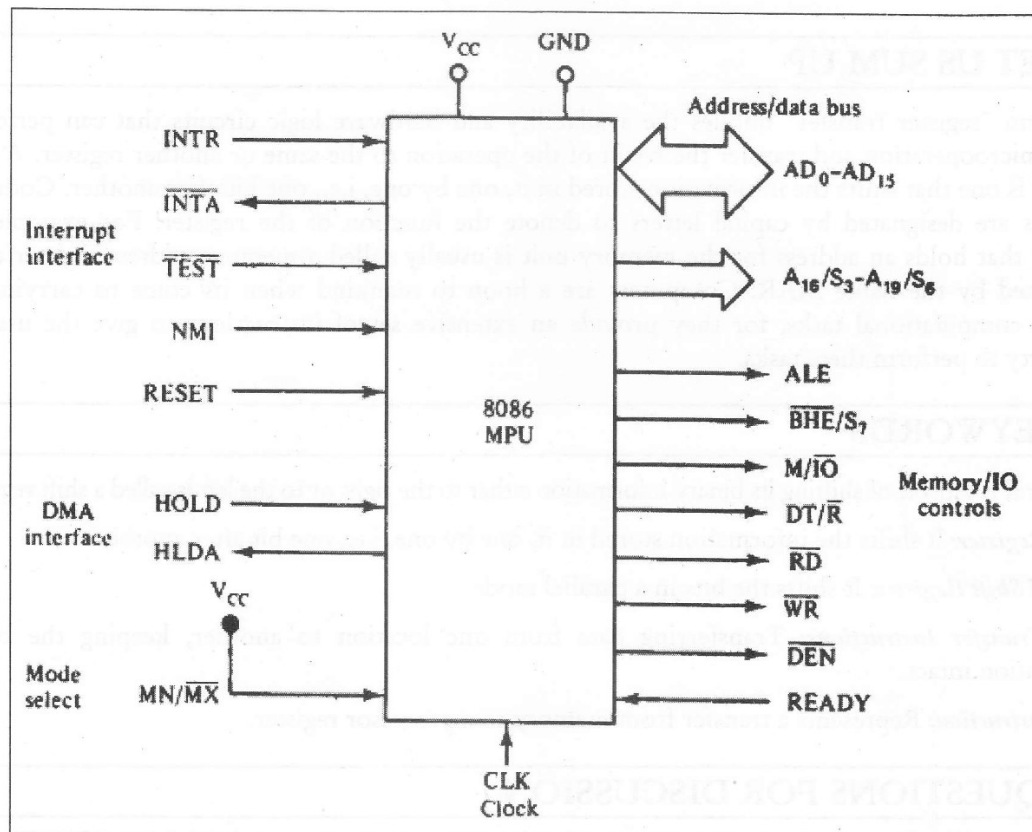


Figure 5.8: Block Diagram of Minimum Mode 8086 MPU

Check Your Progress

Fill in the blanks:

1. The sequence of .... performed on the binary information stored in the registers.
2. A serial register is one that shifts the .... stored in it, one by one, i.e., one bit after another.
3. A parallel load control to enable parallel .... and the n - input lines associated with the parallel transfer.
4. The register that holds an address for the memory unit is usually called a .... address register.
5. The move instruction is employed in computers with multiple .... registers to designate a transfer from one register to another.

5.8 LET US SUM UP

The term "register transfer" implies the availability and hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. A serial register is one that shifts the information stored in it, one by one, i.e., one bit after another. Computer registers are designated by capital letters to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR. Computers are a boon to mankind when its come to carrying out various computational tasks, for they provide an extensive set of instructions to give the user the flexibility to perform these tasks.

5.9 KEYWORDS

Register: It is capable of shifting its binary information either to the right or to the left is called a shift register.

Serial Register: It shifts the information stored in it, one by one, i.e., one bit after another.

Parallel Shift Register: It shifts the bits in a parallel mode.

Data Transfer Instructions: Transferring data from one location to another, keeping the binary information intact.

Load Instruction: Represents a transfer from memory to a processor register.

5.10 QUESTIONS FOR DISCUSSION

1. What is the difference between Register Transfer and Data Transfer Instruction?
2. Explain the four types of Shift Registers.
3. Explain in how many ways data manipulation instructions can be classified.
4. Discuss the organization of Microprocessor 8086.

Check Your Progress: Model Answers

1. Microoperations
2. Information
3. Transfer
4. Memory
5. CPU

5.11 SUGGESTED READINGS

Sajjan G. Shiva; *Computer Design and Architecture*; Marcel Dekker

Silvia Melitta Mueller, Wolfgang J. Paul; *Computer Architecture*; Springer

Joseph D. Dumas II; *Computer Architecture*; CRC Press

Nicholas P. Carter; *Schaum's Outline of Computer Architecture*; Mc. Graw-Hill Professional

UNIT III

LESSON

6

MICRO PROGRAMMING

CONTENTS

6.0 Aims and Objectives

6.1 Introduction

6.2 How Microprogramming Works?

6.2.1 Computer Configuration

6.3 Microoperations

6.4 Arithmetic Microoperations

6.5 Logic Microoperations

6.6 Shift Microoperations

6.7 Let us Sum up

6.8 Keywords

6.9 Questions for Discussion

6.10 Suggested Readings

6.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of micro programming
- Discuss arithmetic micro-operations
- Discuss logic micro-operations
- Define shift micro-operations

6.1 INTRODUCTION

The control parts of computers prior to the mid 60s were constructed, essentially, of electronic components structured into logic gates. It was quickly discovered that building computers, especially the control logic, was complex and error-prone — hence techniques were developed to further structure systems and reduce errors. Nevertheless, it was still difficult and complex, and errors were hard to fix.

In 1957 Maurice Wilkes proposed an alternative called 'microprogramming'. At the time, it was highly impractical, but in 1964 IBM launched its 360 range, most of which used this microprogramming concept. Only the top of the range machine was not microprogrammed, because microprogramming could not deliver the required performance (traditional methods, for all their faults, were fast).

The 360 series were highly significant machines in the 60s, and their influence on machine architecture design is still visible today, particularly in Intel 80x86 and Motorola 680x0 processors, whose instruction sets are essentially evolutions of the 360 original. In fact, IBM still produces mainframes today that use the same architecture.

Wilkes' idea was that each machine instruction was divided into a number of sub-instructions, or microinstructions. While a real instruction (we might say macroinstruction) might be something like 'add the contents of registers A and B', a microinstruction might be something like 'write out register A to bus Z', or 'read data bus into register X' — very basic actions that could be assembled to implement the actual instruction set of the machine. The set of microinstructions that made up a full instruction set was called the microprogram, or microcode.

6.2 HOW MICROPROGRAMMING WORKS?

The precise details of Wilkes' design are not important to us, but we note the following:

- It was based on a memory consisting of a network of diodes.
- It was far too expensive at the time it was suggested, because of the size of the diode array required.
- The arrival of cheap *ferrite core* memory, and then semi-conductor memory, changed this and made the idea viable.
- Even so, unfortunately, the scheme as proposed by Wilkes would not work for technical reasons, and had to be modified slightly.

The idea is that each microinstruction will be divided up into two parts — the control part, which controls the operation of the data path (or data unit), and the address part, which is the address of the next microinstruction to be executed under certain conditions.

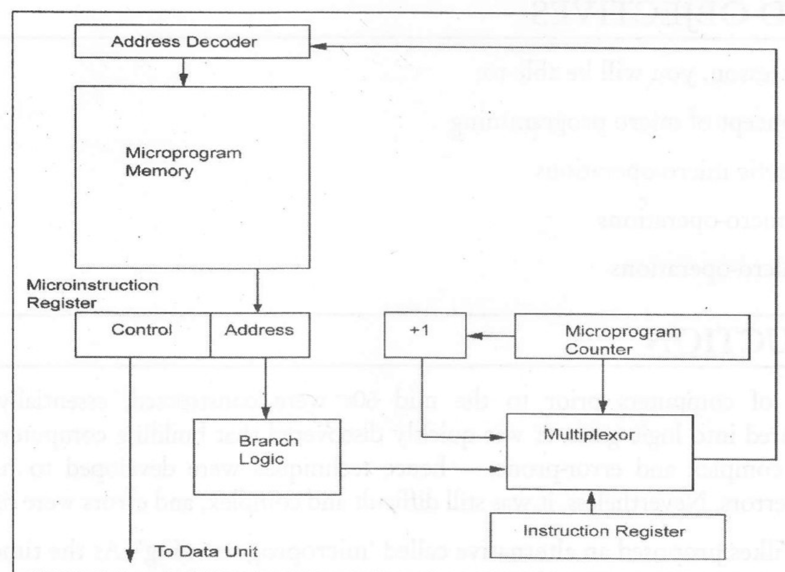


Figure 6.1: A Microgrammed Controller

Consider Figure 6.1. After executing a particular microinstruction, the next microinstruction to be executed is *either* that specified by the address field of the current microinstruction, or the *next sequential* microinstruction (as in a conventional machine language) depending on the value of the conditional inputs. In addition, when one sequence of microinstructions has finished, the next is determined by the contents of the machine instruction register.

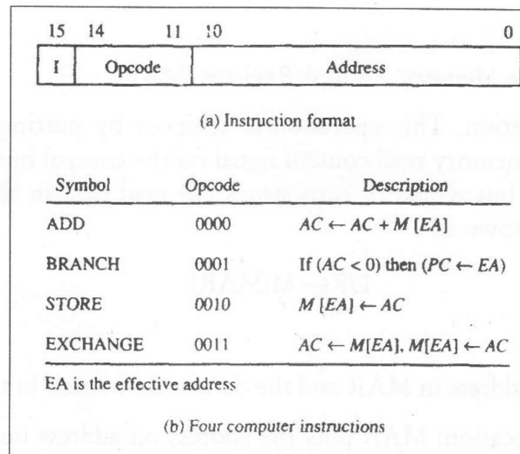
In fact, the situation is usually slightly more complex than this — there is usually a *sequencing* part to the microinstruction in addition to the control and address parts. The sequencing information is used, for example, to decide which of several conditional inputs is to be used to control branching. Usually we require unconditional branching as well, and also many microinstructions do not require branching at all.

6.2.1 Computer Configuration

The processor registers are program counter PC, address register AR, data register DR and accumulator register AC. The function of these registers is similar to the basic computer introduced earlier. The control unit has a control address register CAR and a subroutine register SBR.

The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR. PC can receive information only from AR. The arithmetic, logic, and shift unit perform microoperations with data from AC and DR and places the result in AC. Note that memory receives its address from AR. Input data written to memory or read from memory only through DR.

The computer instruction format is depicted in Figure 6.2(a). It consists of three fields- a 1·bit field for indirect address field. Figure 6.2(b) lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.



(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

Figure 6.2: Computer Instructinons

6.3 MICROOPERATIONS

A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations which transfer binary information from one register to another.
2. Arithmetic microoperations which perform arithmetic operations on numeric data stored in registers.
3. Logic microoperations which perform bit manipulation operation on non-numeric data stored in registers.
4. Shift microoperations which perform shift operations on data stored in registers.

These microoperations, as the name suggest, transfer information from one register to another. The information does not change during this microoperation. A register transferred microoperation may be designed as:

$$R_1 \leftarrow R_2$$

which implies that transfer the content of register R_2 to register R_1. The destination register should have a parallel load capability as we expect the register transfer to occur in a predetermined control condition. A common path for connecting various registers is through a common internal data bus of the processor. In general, the size of this data bus should be equal to the number of bits in a general register. There are some transfers which do not take place through the internal data bus, but through the system bus. These transfers are related to memory and input/output modules. Also the input/output operation is treated as a separate activity where normally a program and therefore instructions are executed. Memory transfer is the most important transfer for instruction execution as it has to take place at least once for every instructions.

Memory Transfer

Memory transfer is achieved via a system bus. Since the main memory is a random access memory, therefore, address of the location which is to be used is to be supplied. This address is supplied by the CPU on the address bus. There are two memory transfer operations: Read and Write. Let us consider the CPU structure then the two memory operations will be performed as:

Memory Read

1. Put memory address in the Memory Address Register (MAR).
2. Read the data of the location. This operation is achieved by putting the MAR in the data on address bus along with a memory read control signal on the control bus. The resultant of memory read is put into the data bus which in turn stores the read data in the data register (DR). This whole operation can be shown as:

$$DR \leftarrow M(MAR)$$

Memory Write

1. Put the desired memory address in MAR and the data to be written in the DR.
2. Write the data into the location: MAR puts the address on address bus and DR puts the data on data bus to be written into the memory location addressed by MAR.

$$M (MAR) \leftarrow DR$$

Normally, a memory read or write operation requires more clock cycles than a typical register transfer operation.

6.4 ARITHMETIC MICROOPERATIONS

These microoperations perform some basic arithmetic operations on the numeric data stored in the registers. These basic operations may be addition, subtraction, incrementing a number, decrementing a number and arithmetic shift operation. An 'add' microoperation can be specified as:

$$R_3 \leftarrow R_1 + R_2$$

It implies: add the contents of registers R_1 and R_2 and store them in register R_3.

The add operation mentioned above requires three registers along with the addition circuit at the ALU. Subtraction in many machines is implemented through complement and addition operations such as:

$$R_3 \leftarrow R_1 - R_2$$

$$\Rightarrow \quad R_3 \leftarrow R_1 + (\text{2's complement of } R_2)$$

$$\Rightarrow \quad R_3 \leftarrow R_1 + (\text{1's complement of } R_2 + 1)$$

$$\Rightarrow \quad R_3 \leftarrow R_1 + R_2 + 1$$

An increment operation can be symbolized as:

$$R_1 \leftarrow R_1 + 1$$

while a decrement operation can be symbolized as:

$$R_1 \leftarrow R_1 - 1$$

These increment and decrement operations can be implemented by using a combinational circuit or binary up/down counters. In most of the computers multiplication and division are implemented using add/subtract and shift microoperations. If a digital system has implemented division and multiplication by means of combinational circuits then we can call these as the microoperations for that system. An arithmetic circuit is normally implemented using parallel adder circuits. Each of the Multiplexer (MUX) of the given circuit has two select inputs. This four bit circuit takes input of two 4-bit data values and a carry in bit and outputs the four resultant data bits and a carry out bit. With the different input values we can obtain various microoperations.

Equivalent Microoperation	**Microoperation Name**
$R \leftarrow R_1 + R_2$	Add
$R \leftarrow R_1 + R_2 + 1$	Add with carry
$R \leftarrow R_1 + R_2$	Subtract with borrow