NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT309

COURSE TITLE: Computer Architecture

COURSE GUIDE

CIT 309
COMPUTER ARCHITECTURE

COURSE TEAM:

Developer/Writer: Greg Onwodi
National Open University of Nigeria

Course Coordinator: Rele Afolorunsho
National Open University of Nigeria

Course Editor: Engr. C. Obi

Programme Leader : Prof. Afolabi Adebanjo
National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
No. 5 Dares Salaam Street Off Aminu Kano Crescent Wuse II, Abuja
Nigeria

e-mail: central info@nou.edu.ng URL: www.nou.edu.ng
Published By:
National Open University of Nigeria
Printed 2009
Reviewed and Reprinted 2020

ISBN:

CONTENTS

MODULE 5 OPERATING SYSTEM SUPPORT ERROR DETECTION AND ERROR CORRECTION CODING

UNIT 1 OPERATING SYSTEM OVERVIEW
UNIT 2 SCHEDULING
UNIT 3 MEMORY SYSTEM
UNIT 4 CACHE MEMORY

MODULE DIGITAL LOGIC

UNIT 1 BOOLEAN ALGEBRA
UNIT 2 LOGIC OPERATIONS
UNIT 3 COMBINATIONAL CIRCUITS

MODULE 1: ORGANIZATION AND ARCHITECTURE

UNIT 1: COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT 2: INSTRUCTION SETS CHARACTERISTICS AND FUNCTIONS
UNIT 3: TYPES OF OPERANDS

UNIT 1

1.0 INTRODUCTION

In spite of the variety and pace of change in the computer field, certain fundamental concepts apply consistently throughout. To be sure, the application of these concepts depends on the current state of technology and the price/ performance objectives of the designer.

Many computer manufacturers offer a family of computer models, all with the same architecture but with differences in organization. In a class of computers called microcomputers, the relationship between the architecture and organization is very close. Changes in technology not only influence organization but also result in the introduction of more powerful and more complex architecture. However, because a computer organization must be designed o implement a particular architectural specification, a thorough treatment of organization requires a detailed examination of architecture as well.

2.0 OBJECTIVES

At the end of this unit you should be able to:

- Explain the operational units of a computer system.
- Outline types of operands and operations specific by machine instruction.
- Explain opcodes, operands and addressing modes

3.0 MAIN CONTENT

3.1 COMPUTER ORGANIZATION AND ARCHITECTURE

Although it is difficult to give precise definition, a consensus exists about the general area covered by it. Computer organization refers to the operational units and their interconnection that realize the architectural specification. Examples of architectural attributes include the instruction set, the number of bit used to represent various data types (e. g numbers, characters), I/O mechanism, and techniques for addressing memory. Organizational attributes include those hardware details transparent to the programmer, such as control signals; interfaces between the computer peripherals and memory technology used.

3.2 STRUCTURE AND FUNCTION

A computer is a computer system, contemporary computers contain millions of elementary electronic components.

- **Structure:** The way in which the components are interrelated.
- **Function:** The operation of each individual component as part of the structure.

In term of description, there are two choices: starting at the bottom and building up to a complete description, or beginning with a top view and

decomposing the system into its subparts. Evidence from a number of fields suggest that the top down approach is the clearest and most effective.
The approach taken is that the computer be described from the top down.

Both the structure and functioning of a computer are simple. Figure 1.1 depicts the basic functions that a computer can perform. In general terms, there are only four:

- Data processing
- Data storage
- Data movement
- Control

The computer of course, must be able to process data. The data may take a wide variety of forms, and the range of processing requirements id broad. It is also essential that a computer store data. Even if the computer is processing data on the fly (i.e data come in and get processed and the results go out immediately) the computer must temporarily store at least. Those pieces of data that are being worked on at any given moment. Files of data are stored on the computer for subsequent retrieval and update.

The computer must be able to move data between itself an outside world. The computers operating environment consist of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process is known as input- output (I/O), and the device is referred to as a peripheral. When data are moved over longer distances, to or form a remote device, the process is known as data communications. Finally, there must be control of these three functions. Ultimately, this control is exercised by the individuals who provide the computer with instructions. Within the computer a control unit manages the computers resources and orchestrates the performance of its functional parts in response to those instructions.

There are four main structural components

- **Central processing unit (CPU**): Controls the operations of the computer and performs its data processing functions; often simply referred to as processor.
- Main memory: Stores data
- **I/O:** Moves data between the computer and its external environment.
- **System interconnections**: Some mechanism that provides for communication among CPU, main memory and I/O. A common example of system interconnection is by means of a system bus, consisting of a number of conducting wires to which all the other components attach.

However, the most interesting and complex component is the C. P. U. Its major structural components are as follows:

- **Control unit**: Controls the operations of the CPU and hence the computer.
- **Arithmetic and logic unit (ALU**): Performs the computer data processing functions.
- Registers: Provides storage internal to the CPU.
- **CPU interconnection**: Some mechanism that provides for communication among the control unit, ALU and registers.

COMPUTER COMPONENTS

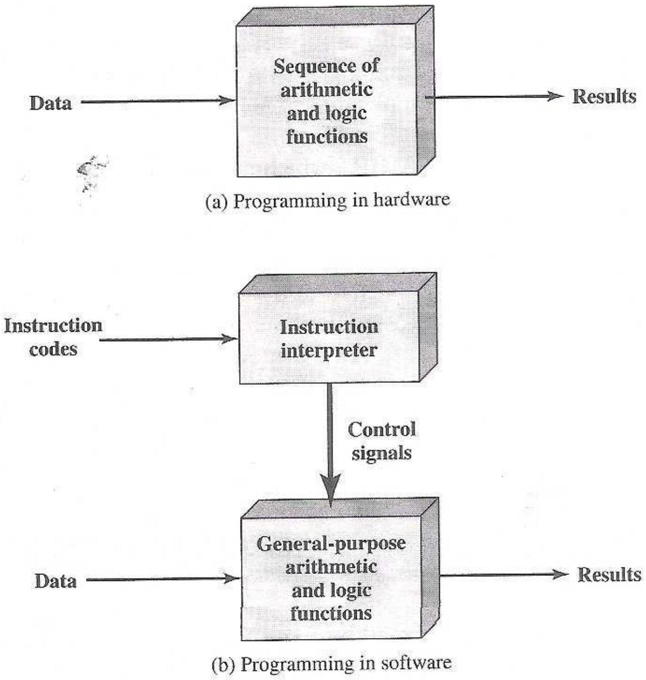
As discussed in Chapter 2, virtually all contemporary computer designs are based on concepts developed by John yon Neumann at the Institute for Advanced Studies Princeton. Such a design is referred to as the ***yon Neumann architecture*** and is base on three key concepts:

- Data and instructions are stored in a single read-write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.
- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

The reasoning behind these concepts was discussed in Chapter 2 but is worth summarizing here. There is a small set of basic logic components that can be combined in various ways to store binary data and to perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program" is in the form of hardware and is termed a *hardwired program.*

Now consider this alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (Figure 3.1a). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.

How shall control signals be supplied? The answer is simple but subtle. The entire program is actually a sequence of steps. At each step, some arithmetic or logical


Data
Sequence of arithmetic and logic functions
Results
Instruction codes
Instruction interpreter
Control signals
Data
General-purpose arithmetic and logic functions
Results

(a) Programming in hardware

(b) Programming in software

Figure 3.1 Hardware and Software Approaches

operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals (Figure 3.1b).

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called *software.*

Figure 3.1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU. Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data anc instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form o an output module. Taken together, these are referred to as I10 *components.*

One more component is needed. An input device will bring instructions an` data in sequentially. But a program is not invariably executed sequentially; it ma,. jump around (e.g., the IAS jump instruction). Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence Thus, there must be a place to store temporarily both instructions and data. That module is called *memory,* or *main memory* to distinguish it from external storage of peripheral devices. Von Neumann pointed out that the same memory could be uses to store both instructions and data.

Figure 3.2 illustrates these top-level components and suggests the interaction, among them. The CPU exchanges data with memory. For this purpose, it typical" makes use of two internal (to the CPU) registers: a memory address register (MAR), which specifies the address in memory for the next read or write, and memory buffer register (MBR), which contains the data to be written into memory receives the data read from memory. Similarly, an I/0 address register (I/OAR specifies a particular 1/0 device. An I/0 buffer (I/OBR) register is used for the $_{ex}$change of data between an I/0 module and the CPU.

A memory module consists of a set of locations, defined by sequentially nun bered addresses. Each location contains a binary number that can be interpreted either an instruction or data. An 1/0 module transfers data from external devices CPU and memory, and vice versa. It contains internal buffers for temporarily holing these data until they can be sent on.

Having looked briefly at these major components, we now turn to an over view of how these components function together to execute programs.

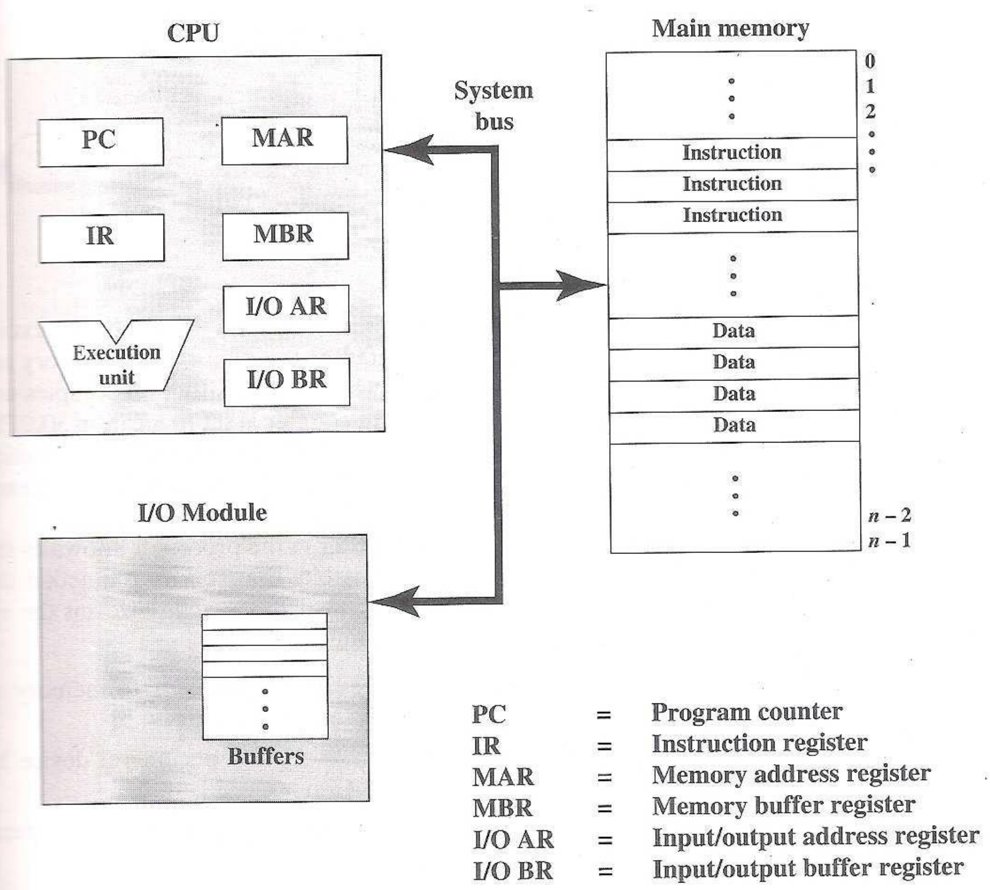


Figure 3.2 Computer Components:Top-Level View

the key elements of program execution. In its simplest form, instruction processing

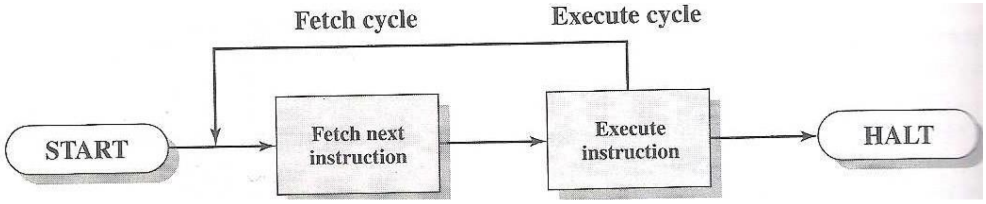
the key elements of program execution. In its simplest form, instruction processing consists of two steps: The processor reads *(fetches)* instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction (see, for example, the lower portion of Figure 2.4).

The processing required for a single instruction is called an *instruction cycle.* Using the simplified two-step description given previously, the

instruction cycle is depicted in Figure 3.3. The two steps are referred to as the *fetch cycle* and the *execute*
cycle. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

INSTRUCTION FETCH AND EXECUTE

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor



Basic Instruction Cycle

always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.

The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

Processor-memory: Data may be transferred from processor to memory or from memory to processor.

Processor-I/O: Data may be transferred to or from a peripheral device be transferring between the processor and an I/O module.

Data processing: The processor may perform some arithmetic or logic operation on data.

Control: An instruction may specify that the sequence of execution be altered For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.

An instruction's execution may involve a combination of these actions.

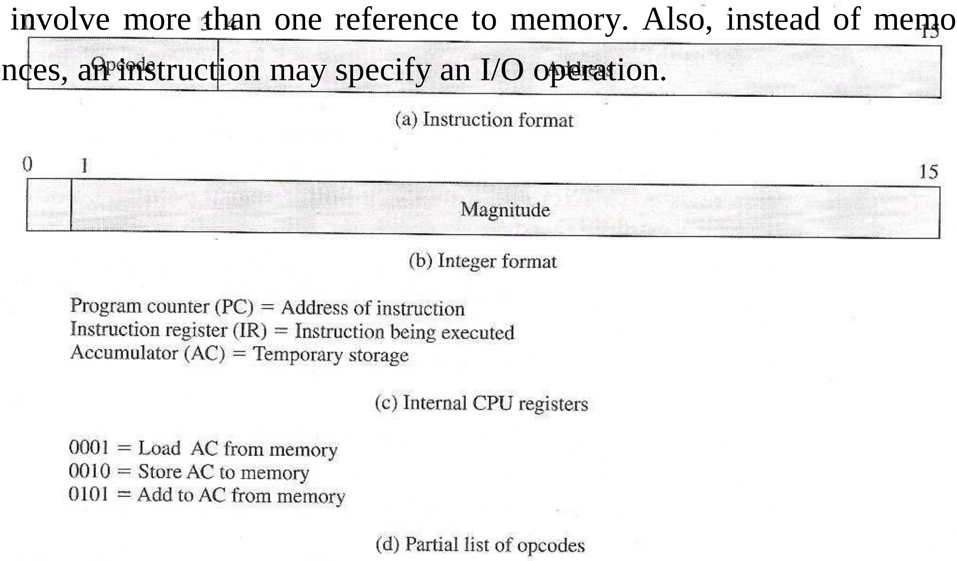
The processor contains a single data register called an accumulator (AC). Both instructions and data are 16 bits long. Thus, it convenient to organize memory using 16-bit words. The instruction format provide 4 bits for the opcode, so that there can be as many as 2^4 = 16 different opcodes, aup to 212 = 4096 (4K) words of memory can be directly addressed.

address 941 and stores the result in the latter location. Three instructions, which be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction value 1940 in hexadecimal) is loaded into the instruction register IR anPC is incremented. Note that this process involves the use of a memory dress register (MAR) and a memory buffer register (MBR). For simply these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is -loaded. The remaining 12 bits (three hexadecimal digits) specify the ac (940) from which data are to be loaded.

3. The next instruction (5941) is fetched from location 301 and the incremented.
4. The old contents of the AC and the contents of location 941 are added an result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the F incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch cycle execute cycle, are needed to add the contents of location 940 to the contents C With a more complex set of instructions, fewer cycles would be needed. Some processors, for example, included instructions that contain more than one address. Thus the execution cycle for a particular instruction on such prop could involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.



(a) Instruction format

(b) Integer format

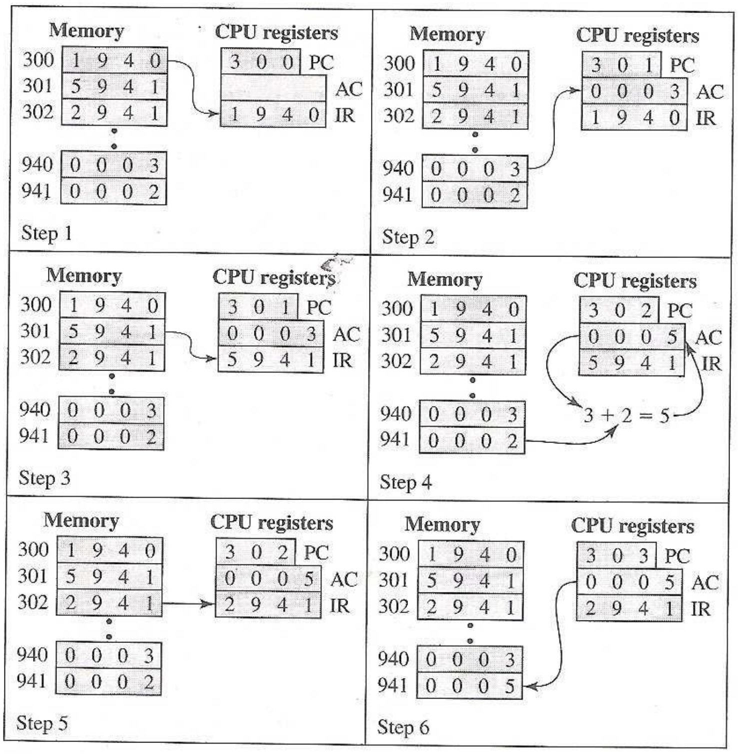
Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 3.4 Characteristics of a Hypothetical Machine



For example, the PDP-11 processor includes an instruction, expressed physically as ADD B,A, that stores the sum of the contents of memory locations B into memory location A. A single instruction cycle with the following steps

- Fetch the ADD instruction.
- Read the contents of memory location A into the processor.
- Read the contents of memory location B into the processor. In order to contents of A are not lost, the processor must have at least two register storing memory values, rather than a single accumulator.
- Add the two values
- Write the result from the processor to memory location A.

Thus, the execution cycle for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instructor specify an I/O operation.

. For any given instruction cycle, some states -null and others may be visited more than once. The states can be described as follows:

Instruction address calculation (ac): Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.

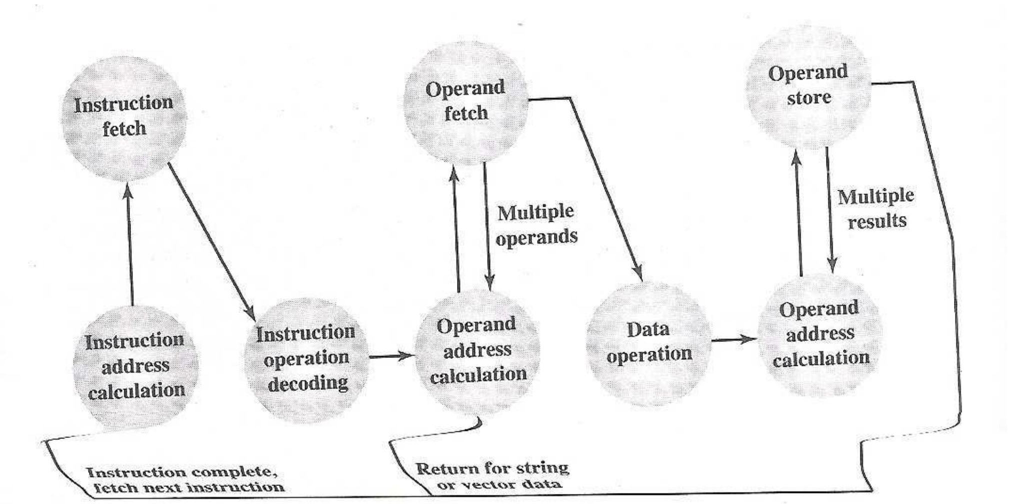

Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

Figure 3.6 Instruction Cycle State Diagram

Instruction fetch (if): Read instruction from its memory location into the processor.

Instruction operation decoding (iod): Analyze instruction to determine type of operation to be performed and operand(s) to be used.

Operand address calculation (oac): If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand. ,

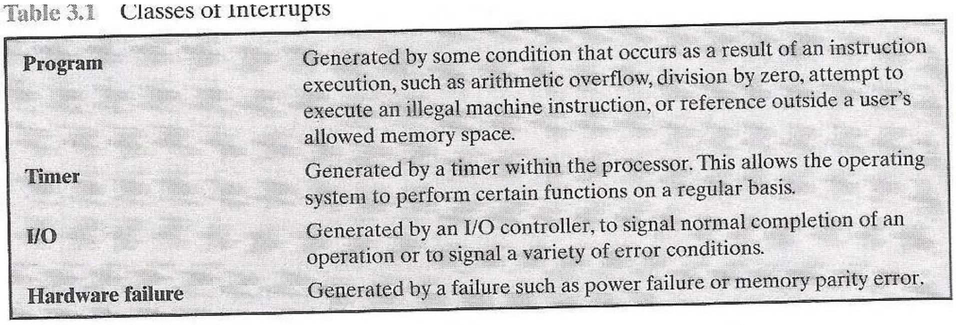
Operand fetch (of): Fetch the operand from memory or read it in from 1/O.

Data operation (do): Perform the operation indicated in the instruction. Operand store (os): Write the result into memory or out to I/O.

States in the upper part of Figure 3.6 involve an exchange between the processor and either memory or an 1/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed. Also note that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this. For example, the PDP-11 instruction ADD A,B results in the following sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.

Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) of characters. As Figure 3.6 indicates, this would involve repetitive operand fetch and/or store operations.

Table 3.1 Classes of Interrupts



Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure such as power failure or memory parity error.

UNIT 2 INSTRUCTION SETS CHARACTERISTICS AND FUNCTIONS

1.0 INTRODUCTION

One boundary where the computer designer and the computer programmer can view the same machine is the machine instruction set. From the designers point of view, the machine instruction set provides the functional requirements for the processor. Implementing the processor is a tasks that in large part involves implementing the machine instruction set.

2.0 OBJECTIVES

At the end ;f the of this unit, you should be able to

Explain the instruction format

Understand the instruction length and characteristics

3.0 MAIN CONTENT

3.1 INSTRUCTION FORMATS

An instruction format defines the layout of the bits of an instruction, in terms of its constituent's fields. An instruction format must include an opcode and implicitly or explicitly, zero or more operands. The format must implicitly explicitly, indicate the addressing mode for each operands. For most instruction sets, more than one instruction format is used.

3.1.1 INSTRUCTION LENGTH

The most basic design issue to be faced is the instruction format length. This decisions effects and is affected by, memory size, memory organization bus structure process complexity and processor speed. This decision determines the richness and flexibility of the machine.

3.2 INSTRUCTION SETS CHARACTERISTICS

The operation of the processor is determined by the instructions it executes referred to as machine instructions or computer instruction. The collection of different instructions that the processor can execute is referred to as the processors instruction set.

3.2.1 ELEMENTS OF A MACHINE INSTRUCTION

These elements are as follows:

- **Operation code:** Specifies the operation to be performed (e.g, ADD, I/O). The operation is specified by a binary code, known as the operation code or opcode.
- **Source operand reference:** This operation may involve one or more source operands, that is operands that are inputs for the operation
- **Results operands reference:** The operation may produce a result
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

The address of the next instruction to be fetched could be either a real address or a virtual address, depending on the architecture. Generally, the distinction is transparent to the instruction set architecture. In most cases, the next instruction to be fetched immediately follows the current instruction. In most cases, there is no explicit reference to the next instruction when an explicit reference is needed then the main memory or virtual memory address must be supplied. Source and result operands can be in one of four areas.

- **Main or virtual memory:** As with next instruction references, the main or virtual memory address must be supplied.
- **Processor register:** With rare exception a processor contains one or more registers that may be referenced by machine instructions. If only one registers exits reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the designed register
- **Immediate:** The value of the operand is contained in a field in the instruction being executed.
- **I/O device:** The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address

3.2.2 INSTRUCTION REPRESENTATION

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituents elements of the instruction

Opcodes are represented by abbreviation called mnemonics that indicate the operation. Common examples include:

ADD	add
SUB	SUBTRACT
MUL	multiply
DIV	divide
LOAD	Load data form memory
STOR	Store data to memory

Operands are also represented symbolically. For example the instruction

ADD, R, Y

May mean add the value contained in data location Y to the contents of register R. In this example Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location not on its address:

Thus, it is possible to write a machine language program in symbolic form.

X= 413

Y= 414

A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions. However symbolic machine language remains a useful toll for describing machine instructions, and we will use it for that purpose.

Lets assume that the variables X and Y corresponds to location 413 and 414.If we assume a simple set of machine instruction, this operation could be accomplished with three instruction.

1. Load a register with the content of memory location 413.
2. Add the contents of memory location 414 to the register.
3. Store the contents of the register in memory location 413.

3.3 INSTRUCTION SET DESIGN

One of the most interesting and most analyzed, aspect of computer design is instruction set is very complex because it affect so many aspect of the computer system. The instruction defines any of the functions performed by the processor and thus has significant effect on the implementation of the process. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set. The most important of these fundamental design issues include the following:

- **Operation repertoire:** How many and which operations to provide and how complex operations should be.
- **Data types:** The various types of data upon which operations are perform.

- Instruction format: Instruction length (in nits) number of assesses size of various fields and so on.
- **Registers:** Number of processor registers that can be referenced by instructions and their use.
- **Addressing:** The mode or modes by which the address of an operand is specified.

These issues are highly interrelated and must be considered together in designing an instruction set.

4.0 CONCLUSION

In spite of the variety and pace of change in the computer field, certain fundamental concept applies consistently throughout. The application of these concepts depends on the current state of technology and the price/performance objectives of the designer.

5.0 SUMMARY

Computer organization refers to the operational units and their interconnections that realize the architectural specification.

Computer architecture refers to those attributes of a system visible to a programmer or those attributes that have a direct impact on the logical execution of a program. Collection of different instruction that the processor can execute is referred to as the processor's instruction set and an to instruction format defines the layout of the bits of an instruction, in terms of its constituents fields.

6.0 TUTOR- MARKED ASSIGNMENT

1. What in general terms is the distinction between computer organization and computer architecture?
2. What are the four main functions of a computer
3. List and briefly explain five important instruction set design issues

7.0 REFERENCES/ FURTHER READING

Sloss, A; symes, D; and Wright, C.ARM system developers guides an Fransisco Morgan Kaufinann, 2004

MODULE 2: Computer Arithmetic

UNIT 1: The arithmetic and logic unit

UNIT 2: Control unit design/Implementation

1.0 INTRODUCTION

This unit focuses on the most complex aspect of the ALU, computer arithmetic. Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point. In both cases, the representation chosen is a crucial design issue and is treated first.

2.0 OBJECTIVES

At the end of this unit, you should be able to

- Understand the way in which numbers are represented (the binary format) and the algorithms used for the basic arithmetic operations (add, subtract, multiply, divide) both to integer and floating point arithmetic.

3.1 THE ARITHMETIC AND LOGIC UNIT

The arithmetic and logic unit (ALU) is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system- Control unit, registers memory, I/0- are there mainly to bring into the ALU for it to process and then take the result back out.

An ALU and all electronic components in the computers are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Figure 3. 1. 1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Data are presented to the ALU in registers and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU. The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored. The flag values are also stored in registers within the processor. The control unit provides signals that control the operation of th ALU and the movement of the data into and out of the ALU.

3.2 INTEGER REPRESENTATION

In the binary number, arbitrary numbers can be represented with just the digits zero and one the minis sign and the period or radix point.

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however w do not have the benefits of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to non negative integers, the representation is straight forward.

An 8 bit word can represent the numbers form 0 to 255, including

00000000	=	0
00000001	=	1
00101001	=	41
10000000	=	128
11111111	=	255

In general, if an n- bit sequence of binary digits is interpreted as an unsigned integer, A it value is

A= n-1
Σ2i ai
2=0

In going from the first to the second equation, we require that the least significant *n* - 1 bits do not change between the two representations. Then we get to .-next to last equation, which is only true if all of the bits in positions throem *2* are 1. Therefore, the sign-extension rule works. .

Fixed-point representation

Finally, we mention that the representations discussed in this section are sometime referred to as fixed point. This is because the radix point (binary point) is fixed assumed to be to the right of the rightmost digit. The programmer can use the representation for binary fractions by scaling the numbers so that the binary poor implicitly positioned at some other location.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.

Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

bitwise complement

As expected, the negative of the negative of that number is itself:

Again, interpret an n-bit sequence of binary digits *a,-Ian-2 ...* a_la_o as a twos complement integer A, so that its value is

Now form the bitwisc complement a, and, treating this is an unsigned integer, add 1. Finally, interpret the resulting n-bit sequence of binary digits as a twos complement integer *B,*so that its value is

A=-2n-1a n-1 +

Some such anomaly is unavoidable. The number of different bit patterns in arn-bit word is 2n, which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is on=_ one representation of 0 (twos complement), then there must be an unequal numb -- - of negative and positive numbers represented..In the case of twos complement, for such an n-bit length, there is a representation for

$^{e-}$~but not for +2$^{''-1}$.

Addition in twos complement is illustrated in Figure 9.3. Addition proceeds as it :_ two numbers were unsigned integers. The first four examples illustrate sucreoperations. If the result of the operation is positive, we get a positive number in :-.- - complement form, which is the same as in unsigned-integer form. If the result o= : - _ operation is negative, we get a negative number in twos complement form. N - that, in some instances, there is a carry bit beyond the end of the word (indicat-$^{-}$ - shading), which is ignored.

On any addition, the result may be larger than can be held in the wor- - i being used. This condition is called overflow. When overflow occurs, the ALL -- _ signal this fact so that no attempt is made to use the result. To detect overflo-. following rule is observed:

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in Figure 9.5. The circle in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive *k* (or subtract negative k), to that number by moving *k* positions clockwise, and we can subtract positive *k (of* add negative *k)* from that number by moving *k* positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; I = overflow). For subtraction, the

4.0 CONCLUSION

Numbers are represented in binary form and the algorithms used for basic arithmetic operators are add, subtract, multiply and divide

5.0 SUMMARY

- An ALU and all electronic components in the digital logic devices that store binary digits and perform simple Boolean logic operations

- Overflow rule occurs when two numbers positive or negative numbers are added and the result of the addition has the opposite sign.
- Subtraction flow is to subtract one number (subtracted) from another (minuend) take the two compliments (negation) of the subtrahend and hold it to the minuend.

Floating point numbers are expressed as a number (significant) multiplied by a constant (base) raised to some integer power (exponent). It can be used to represent very large and very small numbers.

6.0 TUTOR- MARKED ASSIGNMENT

1. What is sign- extension rule for twos compliment numbers?
2. Find the following differences using two compliment arithmetic:

a. 1111011 b. 10101110 c. 111110010111
-100100 -111-1-1 -111010010101

7.0 Reference and further reading

Swartzlander, E. editor computer Arithimetic, volumes I and II. Los Alamitiss, CA IEEE Computer society press, 1990.

UNIT 2: CONTROL UNIT DESIGN/OPERATION

7.0 Reference and further reading

1.0 Introduction

The execution of an instruction involves the execution of a sequence of sub steps, generally called cycles. For example an execution may consist of fetch, indirect, execute and interrupt cycles. Each cycle is in turn made up is a sequence of more fundamental operations called micro- operations. A single micro operation generally involves a transfer between registers a transfer between registers a register and an external bus, or a simple ALU operation.

2.0 At the end of this unit you should be able to

- Understand that each cycle is in turn made up of a sequence of more fundamental operations called micro- operations.
- Identify the two task performs by the control unit of a processor which are: Generation of control signals that causes each Micro operation to be executed and causing the processor to step through a series of micro- operations in the proper sequence based on the program being executed.

3.1 MICRO OPERATIONS

The prefix micro refers to the fact that each step is very simple and accomplishes very little. To design a control unit each of the smaller cycles involves a series of step each of which involves the processor registers. We refer to these steps as micro operations. Micro operations are the functional, or atomic operations of a processor.

Three. Now, we turn to the question of how these functions are performed or, more specifically, how the various elements of the processor are controlled to provide these functions. Thus, we turn to a discussion of the control unit, which controls the operation of the processor.

We have seen that the operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle. Of course, we must remember that this sequence of instruction cycles is not necessarily the same as the *written sequence* of instructions that make up the program, because of the existence of branching instructions. What we are referring to here is the execution *time sequence* of instructions.

We have further seen that each instruction cycle is made up of a number of smaller units. One subdivision that we found convenient is fetch, indirect, execute, and interrupt, with only fetch and execute cycles always occurring.

To design a control unit, however, we need to break down the description further. In our discussion of pipelining in Chapter 12, we began to see that a further decomposition is possible. In fact, we will see that each of the smaller cycles involves
a series of steps, each of which involves the processor registers. We will refer to these steps as micro-operations. The prefix *micro* refers to the fact that each step is very simple and accomplishes very little. Figure 15.1 depicts the relationship among the various concepts we have been discussing. To summarize, the execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter subcycles (e.g., fetch, indirect, execute, interrupt). The execution of each subcycle involves one or more shorter operations, that is, micro-operations.

Micro-operations are the functional, or atomic, operations of a processor. In this section, we will. examine micro-operations to gain an understanding of how

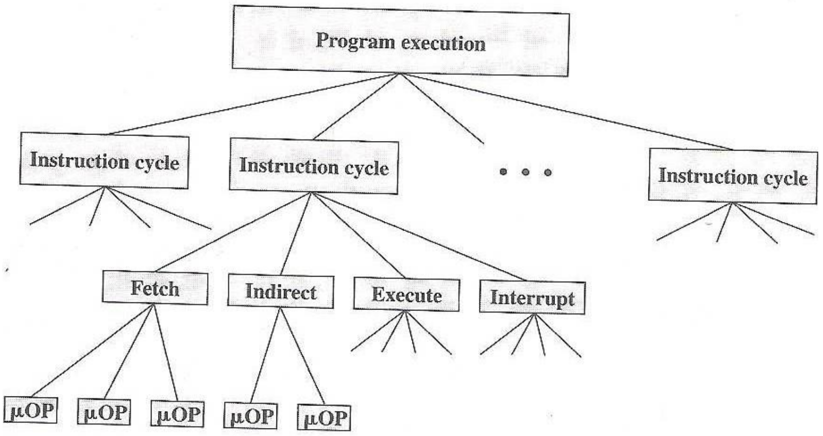


Figure 15.1 Constituent Elements of a Program Execution

the events of any instruction cycle can be described as a sequence of such m' operations. A simple example will be used. In the remainder of this chapter. -then show how the concept of micro-operations serves as a guide to the desi=the control unit.

'We begin by looking at the fetch cycle, which occurs at the beginning of eac_- _struction cycle and causes an instruction to be fetched from memory. For purp,: of discussion, we assume the organization depicted in Figure 12.6. Four register• involved:

Memory address register (MAR): Is connected to the address lines of the tem bus. It specifies the address in memory for a read or write operation.

Memory buffer register (MBR): Is connected to the data lines of the system -- _ It contains the value to be stored in memory or the last value read from melr _ Program counter (PC): Holds the address of the next instruction to be fete==Instruction register (IR): Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the poin_: view of its effect on the processor registers. An example appears in Figure 15.-. .=_ the beginning of the fetch cycle, the address of the next instruction to be execs:.=-is in the program counter (PC); in this case, the address is 1100100. The first steto move that address to the memory address register

(MAR) because this is only register connected to the address lines of the system bus. The second step bring in the instruction. The desired address (in the MAR) is placed on the ad c== -

We have seen that the operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle. Of course, we must remember that this sequence of instruction cycles is not necessarily the same as the *written sequence* of instructions that make up the program, because of the existence of branching instructions. What we are referring to here is the execution *time sequence* of instructions.

We have further seen that each instruction cycle is made up of a number of smaller units. One subdivision that we found convenient is fetch, indirect, execute, and interrupt, with only fetch and execute cycles always occurring.

To design a control unit, however, we need to break down the description further. In fact, we will see that each of the smaller cycles involves a series of steps, each of which involves the processor registers. We will refer to these steps as micro-operations. The prefix *micro* refers to the fact that each step is very simple and accomplishes very little. Figure 15.1 depicts the relationship among the various concepts we have been discussing. To summarize, the execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter subcycles (e.g., fetch, indirect, execute, interrupt). The execution of each subcycle involves one or more shorter operations, that is, micro-operations.

Micro-operations are the functional, or atomic, operations of a processor.

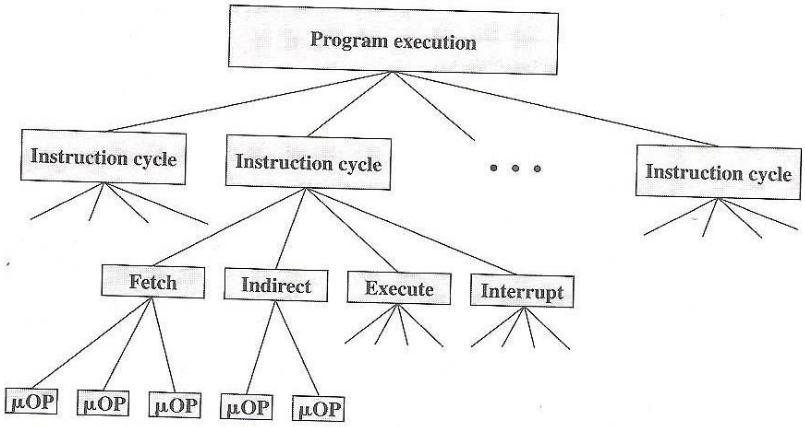


Figure 15.1 Constituent Elements of a Program Execution

bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by the instruction length to get ready for the next instruction. Because these two actions (read word from memory, increment PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

Thus, the simple fetch cycle actually consists of three steps and four microoperations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. Symbolically, we can write this sequence of events as follows:

t_1: MAR E- (PC) t2: MBR <-- Memory PC <- (PC) + I t3: IR <-- (MBR)

where I is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (t_i, t_2, t_3) represents successive time units. In words, we have

I First time unit: Move contents of PC to MAR.

Second time unit: Move contents of memory location specified by MAR to MBR. Increment by I the contents of the PC.

Third time unit: Move contents of MBR to IR.

Note that the second and'third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

t_1: MAR <- (PC) t_2: MBR <- Memory t3: PC E- (PC) + I IR <- (MBR)

The groupings of micro-operations must follow two simple rules:

The proper sequ**en**ce of events must be followed. Thus (MAR - (PC)) must precede (MBR - Memory) because the memory read operation makes use of the address in the MAR.

Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR ¢-- Memory) and (IR <- MBR) should not occur during the same time unit.

A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor. We defer a discussion of this point until later in this chapter.

It is useful to compare events described in this and the following subsections to Figure 3.5. Whereas micro-operations are ignored in that figure, this discussion shows the micro-operations needed to perform the subcycles of the instruction cycle.

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then ar indirect cycle must precede the execute cycle. The data flow differs somewhat from_ that indicated in Figure 12.7 and includes the following micro-operations:

t_l: MAR <-- (IR(Address)) t_2: MBR
F- Memory
t_3: IR(Address) F- (MBR(Address))

The address field of the instruction is transferred to the MAR. This is then use to fetch the address of the operand. Finally, the address field of the IR is update from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been use and it is ready for the execute cycle. We skip that cycle for a moment, to consider t interrupt cycle.

At the completion of the execute cycle, a test is made to determine whether any^{-} :-_abled interrupts have occurred. If so, the interrupt cycle occurs. The nature of cycle varies greatly from one machine to another. We present a very simple sequeof events, as illustrated in Figure 12.8. We have

t_1: MBR E- (PC)
t_2: MAR F- Save Address PC F-
Routine Address t_3: Memory E-
(MBR)

In the first step, the contents of the PC are transferred to the MBR, so that u- can be saved for return from the interrupt. Then the MAR is loaded with the add- .at which the contents of the PC are to be saved, and the PC is loaded with the add to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.

This is not true of the execute cycle. Because of the variety opcodes, there are a number of different sequences of micro-operations that can occur. Let us consider several hypothetical examples.

First, consider an add instruction:

ADD R1, X

which adds the contents of the location X to register R1. The following sequence of micro-operations might occur:

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finallv. the contents of RI and MBR are added by the ALLT. Again. this is a simplified example. Additional micro-operations may be required to extract the register reference from the IR and perhaps to stage the ALt' inputs or outputs in some intermediate registers.

Let us look at two more complex examples. A common instruction is increment and skip if zero:

The content of location X is incremented by l. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

t_i: MAR <-- (IR(address))
t_2: MBR- F- Memory
t_z : MBR <-- (MBR) + 1
t_u: Memory <- (MBR)
If ((MBR) = 0) then (PC F - (PC) + I)

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

It is worth pondering the minimal nature of the control unit. The control unit is the engine that runs the entire computer. It does this based only on knowing the instructions to be executed and the nature of the results of arithmetic and logical operations (e.g., positive, overflow, etc.). It never gets to see the data being processed or the actual results produced. And it controls everything with a few control signals to points within the processor and a few control signals to the system bus.

INTERNAL PROCESSOR ORGANIZATION

Figure 15.5 indicates the use of a variety of data paths. The complexity of this type of organization should be clear. More typically, some sort of internal bus arrangement, as was suggested in Figure 12.2, will be used.

Using an internal processor bus, Figure 15.5 can be rearranged as shown in Figure 15.6. A single internal bus connects the ALU and all processor registers.

CPU with Internal Bus.

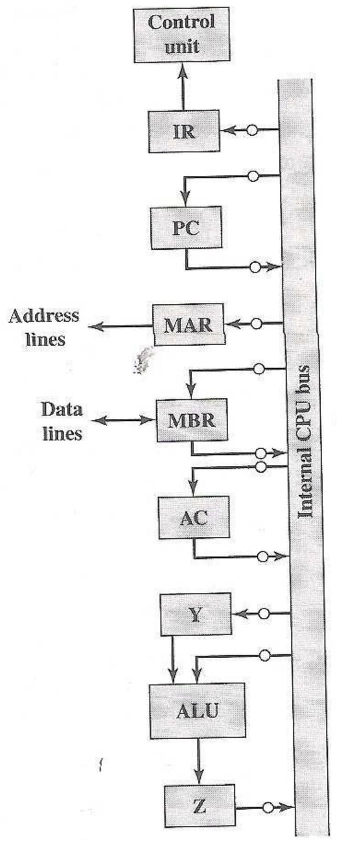


Figure 15.6 CPU with Internal Bus

Gates and control signals are provided for movement of data onto and off the bus from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

Two new registers, labeled Y and Z, have been added to the organization. These are needed for the proper operation of the ALU. When an operation involving two operands is performed, one can be obtained from the internal bus, but the other must be obtained from another source. The AC could be used for this purpose, but this limits the flexibility of the system and would not work with a processor with multiple general-purpose registers. Register Y provides temporary storage for the other input. The ALU is a combinatorial circuit (see Chapter 20) with no internal storage. Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Thus, the output of the ALU cannot be directly connected to the bus, because this output would feed back to the input. Register Z provides temporary output storage. With this arrangement, an operation to add a value from memory to the AC would have the following steps:

t_1: MAR <-- (IR(address)) t_2:
MBR E- Memory
t_3: Y <__ (MBR)
t4: Z f- (AC) +
(Y) t_s: AC F- (Z)

Other organizations are possible, but, in general, some sort of internal bus or set of internal buses is used. The use of common data paths simplifies the interconnection layout and the control of the processor. Another practical reason for the use of an internal bus is to save space.

To illustrate some of the concepts introduced thus far in this chapter, let us consider the Intel 8085. Its organization is shown in Figure 15.7. Several key components that may not be self-explanatory are:

Incrementer/decrementer address latch: Logic that can add 1 to or subtract 1 from the contents of the stack pointer or program counter. This saves time by avoiding the use of the ALU for this purpose.

Interrupt control: This module handles multiple levels of interrupt signals.

Serial I/O control: This module interfaces to devices that communicate 1 bit at a time.

Table 15.2 describes the external signals into and out of the 8085. These are linked to the external system bus. These signals are the interface between the 8085 processor and the rest of the system (Figure 15.8).

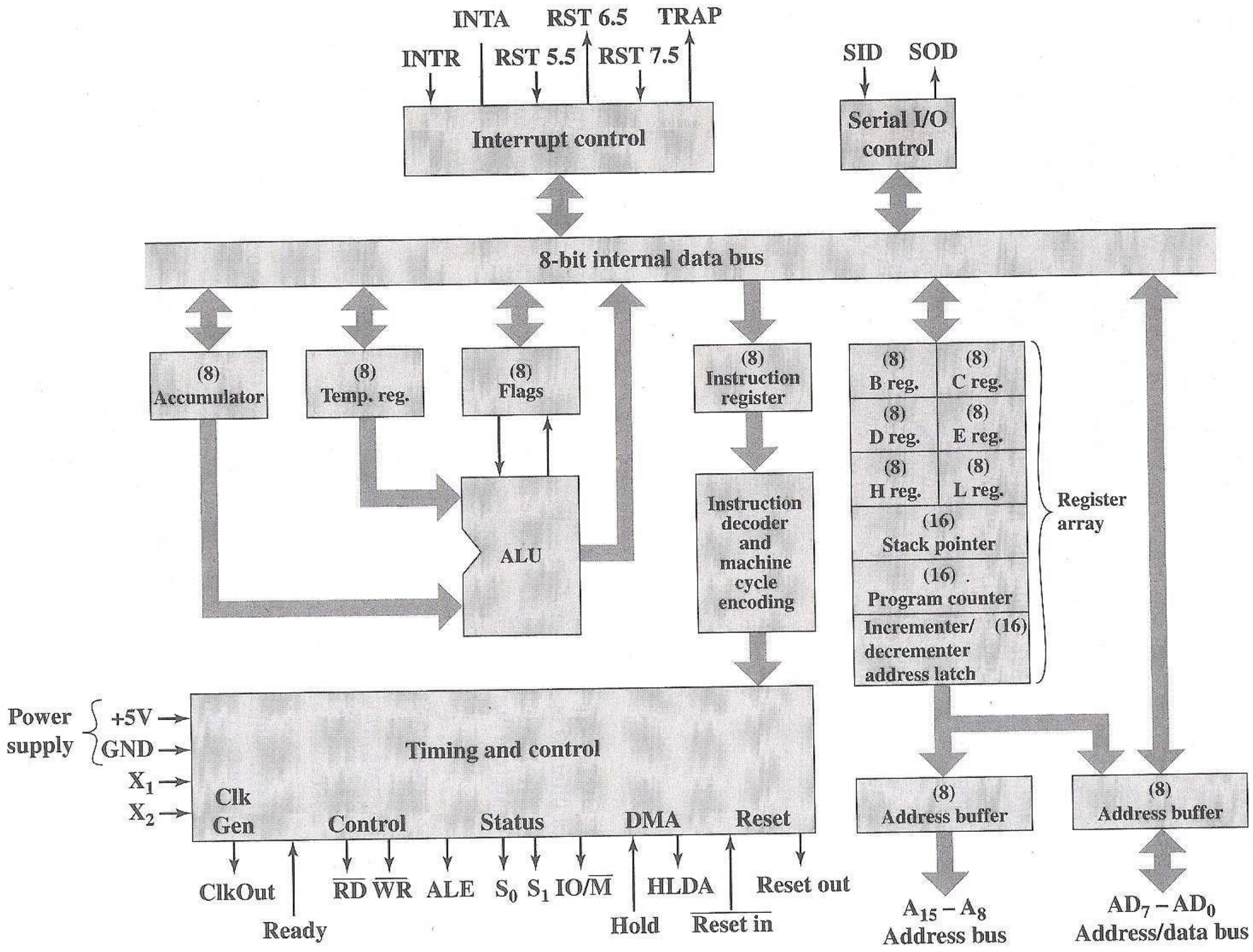


Figure 15.7 Intel 8085 CPU Block Diagram

Table 15.2 Intel 8085 External Signals

Address and Data Signals

High Address (A15–A8)
The high-order 8 bits of a 16-bit address.

Address/Data (AD7–AD0)
The lower-order 8 bits of a 16-bit address or 8 bits of data. This multiplexing saves on pins.

Serial Input Data (SID)
A single-bit input to accommodate devices that transmit serially (one bit at a time).

Serial Output Data (SOD)
A single-bit output to accommodate devices that receive serially.

Timing and Control Signals

CLK (OUT)
The system clock. The CLK signal goes to peripheral chips and synchronizes their timing.

X1, X2
These signals come from an external crystal or other device to drive the internal clock generator.

Address Latch Enabled (ALE)
Occurs during the first clock state of a machine cycle and causes peripheral chips to store the address lines. This allows the address module (e.g., memory, I/O) to recognize that it is being addressed.

Status (S0, S1)
Control signals used to indicate whether a read or write operation is taking place.

IO/M
Used to enable either I/O or memory modules for read and write operations.

Read Control (RD)
Indicates that the selected memory or I/O module is to be read and that the data bus is available for data transfer.

Write Control (WR)
Indicates that data on the data bus is to be written into the selected memory or I/O location.

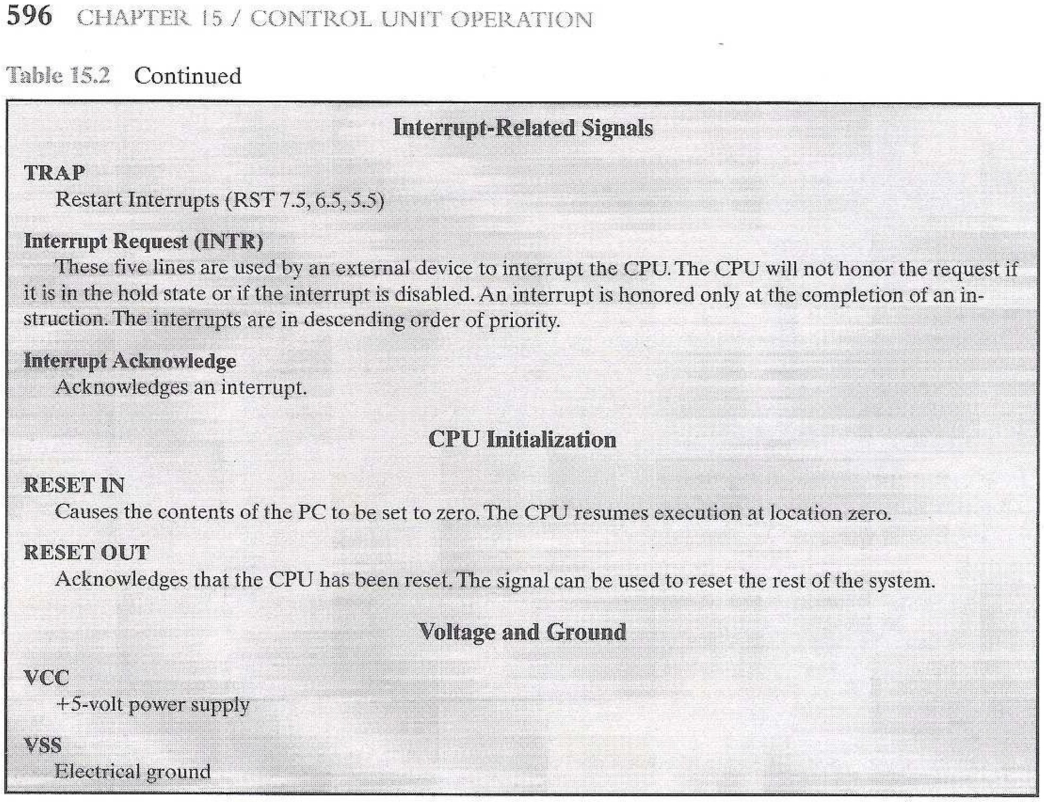
Memory and I/O Initiated Symbols

Hold
Requests the CPU to relinquish control and use of the external system bus. The CPU will complete execution of the instruction presently in the IR and then enter a hold state, during which no signals are inserted by the CPU to the control, address, or data buses. During the hold state, the bus may be used for DMA operations.

Hold Acknowledge (HOLDA)
This control unit output signal acknowledges the HOLD signal and indicates that the bus is now available.

READY
Used to synchronize the CPU with slower memory or I/O devices. When an addressed device asserts READY, the CPU may proceed with an input (DBIN) or output (WR) operation. Otherwise, the CPU enters a wait state until the device is ready.

(Continued)



596 CHAPTER 15 / CONTROL UNIT OPERATION

Table 15.2 Continued

Interrupt-Related Signals

TRAP
Restart Interrupts (RST 7.5, 6.5, 5.5)

Interrupt Request (INTR)
These five lines are used by an external device to interrupt the CPU. The CPU will not honor the request if it is in the hold state or if the interrupt is disabled. An interrupt is honored only at the completion of an instruction. The interrupts are in descending order of priority.

Interrupt Acknowledge
Acknowledges an interrupt.

CPU Initialization

RESET IN
Causes the contents of the PC to be set to zero. The CPU resumes execution at location zero.

RESET OUT
Acknowledges that the CPU has been reset. The signal can be used to reset the rest of the system.

Voltage and Ground

VCC
+5-volt power supply

VSS
Electrical ground

The control unit is identified as having two components labeled (1) in' decoder and machine cycle encoding and (2) timing and control. A discuss"C _-- first component is deferred until the next section. The essence of the contthe timing and control module. This module includes a clock and accepts as i¯ current instruction and some external control signals. Its output consists C¯ : -- signals to the other components of the processor plus control signals to the :-_- system bus.

The timing of processor operations is synchronized by the clock trolled by the control unit with control signals. Each instruction cycle i, into from one to five *machine cycles;* each machine cycle is in turn diN from three to five states. Each state lasts one clock cycle. During a state. the

son performs one or a set of simultaneous micro-operations as determine -

control signals.

The number of machine cycles is fixed for a given instruction but one instruction to another. Machine cycles are defined to be equivalent cesses. Thus, the number of machine cycles for an instruction depends on

bar of times the processor must communicate with external devices. For e an instruction consists of two 8-bit portions, then two machine, cycles are fetch the instruction. If that instruction involves a 1-byte memory or 1/0 then a third machine cycle is required for execution.

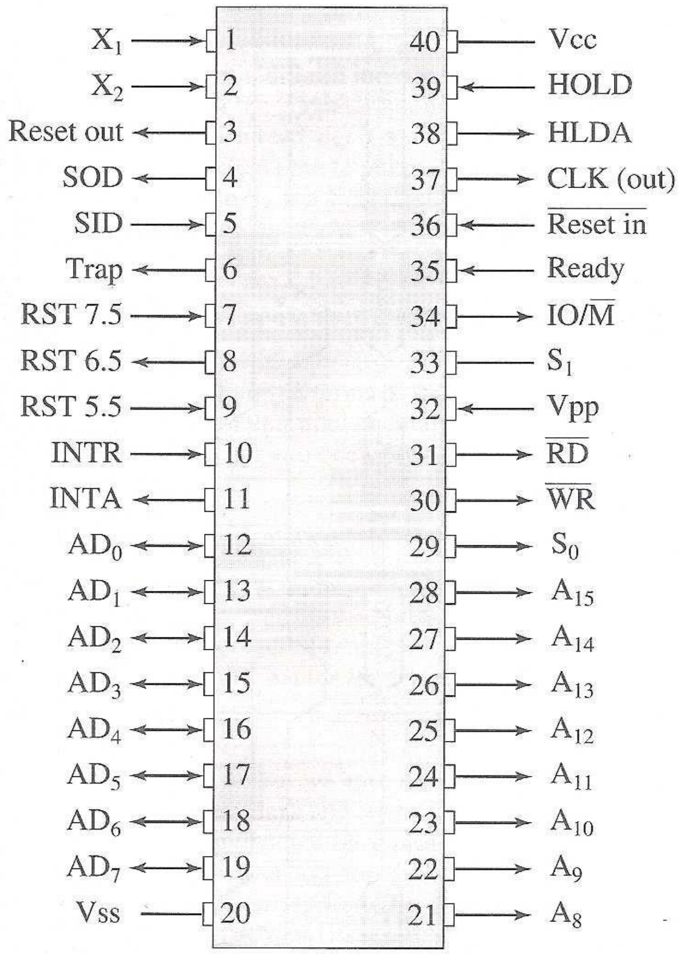


Figure 15.8 Intel 8085 Pin Configuration

Figure 15.9 gives an example of 8085 timing, showing the value of external control signals. Of course, at the same time, the control unit generates internal control signals that control internal data transfers. The diagram shows the instruction cycle for an OUT instruction. Three machine cycles (M_l, M_2, M_3) are needed. During the first, the OUT instruction is fetched. The second machine cycle fetches the second half of the instruction, which contains the number of the 1/O device selected for output. During the third cycle, the contents of the AC are written out to the selected device over the data bus.

The Address Latch Enabled (ALE) pulse signals the start of each machine cycle from the control unit. The ALE pulse alerts external circuits. During timing state T_1 of machine cycle M_r, the control unit sets the IO/M signal to indicate that this is a memory operation. Also, the control unit causes the contents of the PC to be placed on the

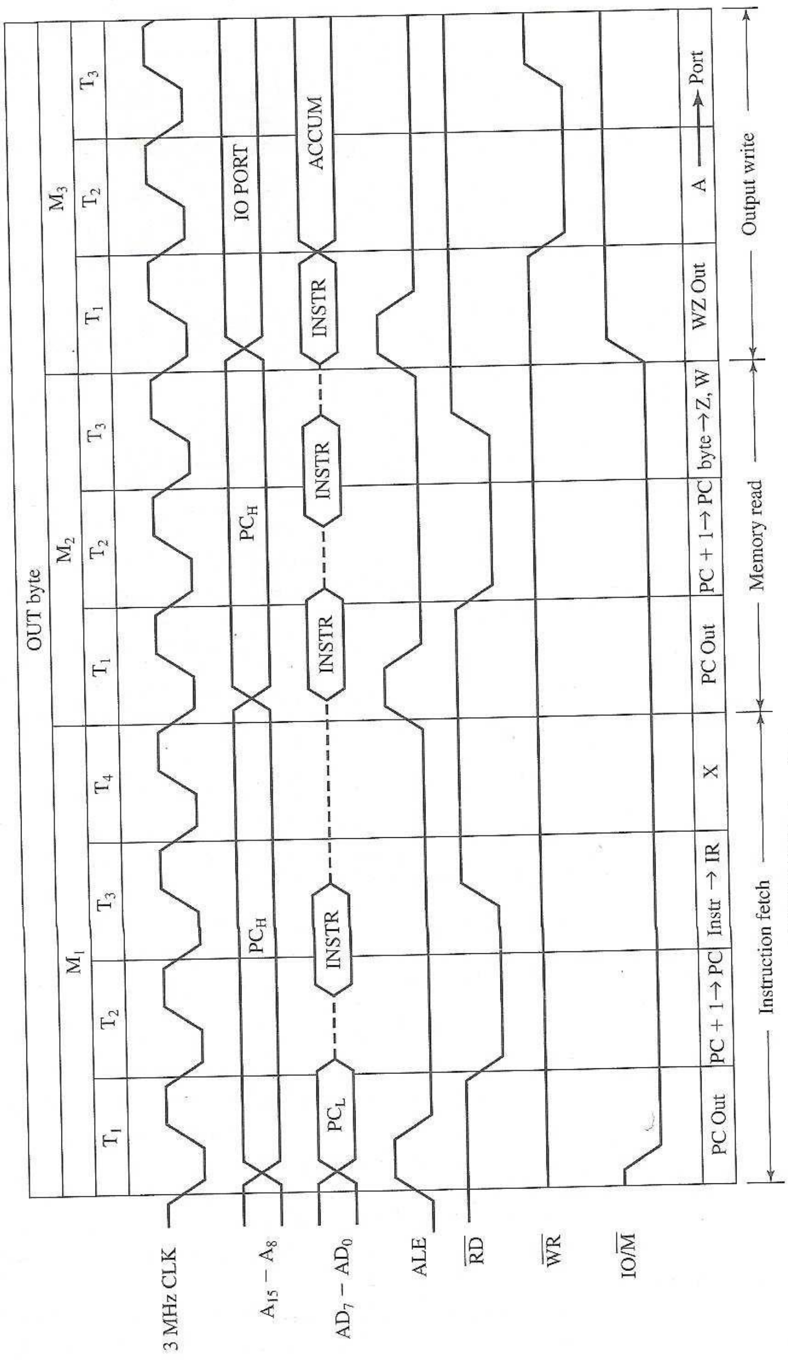


Figure 15.9 Timing Diagram for Intel 8085 OUT Instruction

addressed memory module places the contents of the addressed memory vocation on the address/data bus. The control unit sets the Read Control (RD) signal to indicate a read, but it waits until T_3 to copy the data from the bus. This gives the memory module time to put the data on the bus and for the signal levels to stabilize. The final state, T_4, is a bus *idle* state during which the processor decodes the instruction. The remaining machine cycles proceed in a similar fashion.

Finally, consider a subroutine call instruction. As an example, consider a branchand-save-address instruction:

BSA X

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues at location X + I. The saved address will later be uses for return. This is a straightforward technique for providing subroutine calls. The fo=lowing micro-operations suffice:

t_2: MAR E- (IR(address)) MBR ~-
(PC)
t_z: PC <-- (IR(address)) Memory <--
(MBR) t_3: PC <__ (PC) + I

The address in the PC at the start of the instruction is the address of the nexinstruction in sequence. This is saved at the address designated in the IR. The lateeaddress is also incremented to provide the address of the instruction for the next it - struction cycle.

We have seen that each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations. In our example, there is one sequence eac= for the fetch, indirect, and interrupt cycles, and, for the execute cycle, there is one sequence of micro-operations for each opcode.

To complete the picture, we need to tie sequences of micro-operations together, and this is done in Figure 15.3. We assume a new 2-bit register called the *instruction cycle code (ICC).* The ICC designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch 01: Indirect
10: Execute 11:
Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle (see Figure 12.4). For both the fetch and execute cycles, the next cycle depends on the state of the system.

Thus, the flowchart of Figure 15.3 defines the complete sequence of microoperations, depending only on the instruction sequence and the interrupt pattern. Of course, this is a simplified example. The flowchart for an actual processor would be more complex. In any case, we have reached the point in our discussion in whic?, the operation of the processor is defined as the performance of a sequence of microoperations. We can now consider how the control unit causes this sequence to occur.

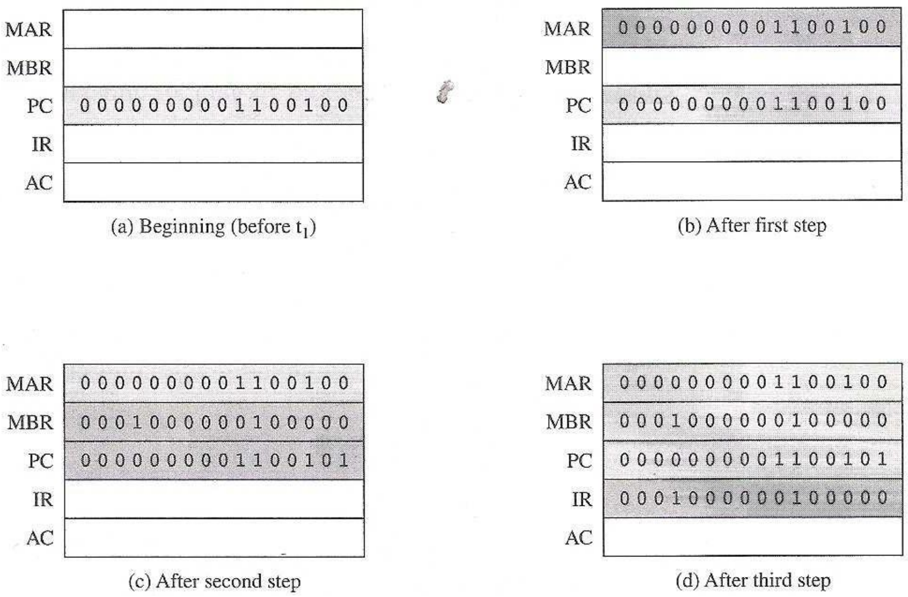
of tbp ~~r of the interrupt-processing routine. These two actions may each be - single micro-operation. However, because most processors provide multiple tyr_ and/or levels of interrupts, it may take one or more additional micro-operations obtain the Save Address and the Routine Address before they can be transfer the events of any instruction cycle can be described as a sequence of such micro operations. A simple example will be used. In the remainder of this chapter, we then show how the concept of micro-operations serves as a guide to the design of the control unit.

THE FETCH CYCLE

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. Four registers are involved:

- **Memory address register (MAR):** Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory buffer register (MBR):** Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
- **Program counter (PC):** Holds the address of the next instruction to be fetched.
- **Instruction register (IR):** Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears in Figure 3.1.2. At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100. The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus. The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address



MAR	
MBR	
PC	0000000001100100
IR	
AC	

(a) Beginning (before t_1)

MAR	0000000001100100
MBR	
PC	0000000001100100
IR	
AC	

(b) After first step

MAR	0000000001100100
MBR	0001000000100000
PC	0000000001100101
IR	
AC	

(c) After second step

MAR	0000000001100100
MBR	0001000000100000
PC	0000000001100101
IR	0001000000100000
AC	

(d) After third step

bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by the instruction length to get ready for the next instruction. Because these two actions (read word from memory, increment PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

Thus, the simple fetch cycle actually consists of three steps and four micro-operations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. Symbolically, we can write this sequence of events as follows:

where I is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (t1, t2, t_3) represents successive time units. In words, we have

- **First time unit:** Move contents of PC to MAR.
- **Second time unit:** Move contents of memory location specified by MAR to MBR. Increment by I the contents of the PC.
- **Third time unit:** Move contents of MBR to IR.

Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

The groupings of micro-operations must follow two simple rules:

The proper sequence of events must be followed. Thus (MAR - (PC)) must precede (MBR - Memory) because the memory read operation makes use of the address in the MAR.

Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR Memory) and (IR E- MBR) should not occur during the same time unit.

A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor.

Whereas micro-operations are ignored in that figure, this discussion shows the micro-operations needed to perform the subcycles of the instruction cycle.

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle.

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We have

t_l: MBR <-- (PC)
t_2: MAR <-- Save Address PC <-- Routine
Address t_3: Memory <-- (MBR)

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may take one or more additional micro-operations to obtain the Save Address and the Routine Address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.

This is not true of the execute cycle. Because of the variety opcodes, there are a number of different sequences of micro-operations that can occur. Let us consider several hypothetical examples.

First, consider an add instruction:

which adds the contents of the location X to register R1. The following sequence of micro-operations might occur:

t_1: MAR <-- (IR(address)) t_2: MBR <--
Memory
t_3: R1 ~- (R1) + (MBR)

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.

Let us look at two more complex examples. A common instruction is increment and skip if zero:

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Finally, consider a subroutine call instruction. As an example, consider a branch and-save-address instruction:

BSA X

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues at location X + I. The saved address will later be used for return. This is a straightforward technique for providing subroutine calls. The following micro-operations suffice:

t 1 : MAR <-- (IR(address))

MBR <-- (PC)

t_z: PC ~__ (IR(address))

Memory - (MBR)

t3: PC ~_ (PC) + I

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in the IR. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

THE INSTRUCTION CYCLE

We have seen that each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations. In our example, there is one sequence each for the fetch, indirect, and interrupt cycles, and, for the execute cycle, there is one sequence of micro-operations for each opcode.

We assume a new 2-bit register called the *instruction cycle code (ICC).* The ICC designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch

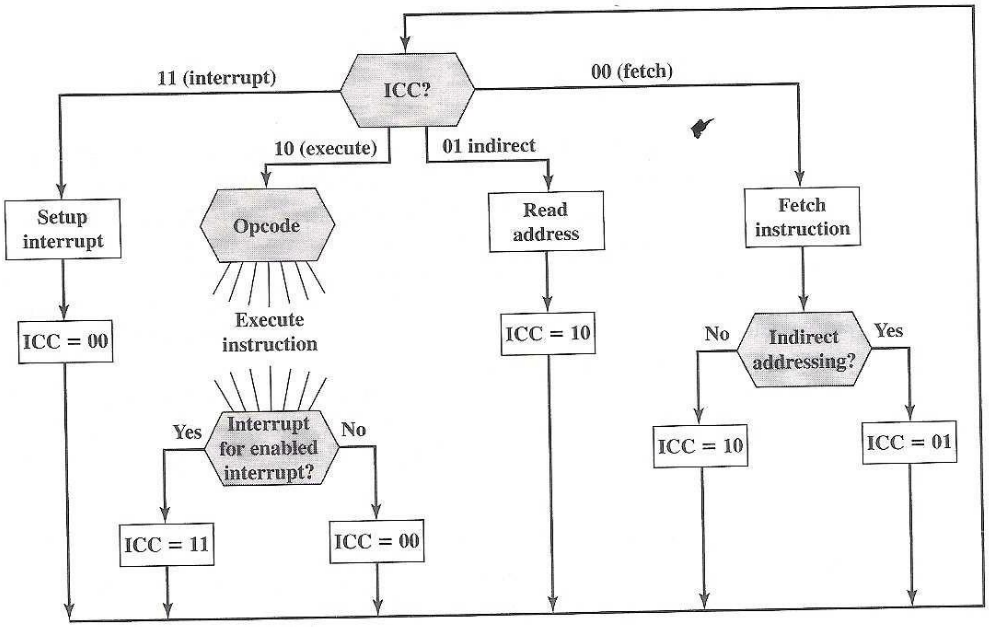
01: Indirect

10: Execute

11: Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle. For both the fetch and execute cycles, the next cycle depends on the state of the system.

Thus, the flowchart of Figure 3.1.3 defines the complete sequence of micro operations, depending only on the instruction sequence and the interrupt pattern. Of course, this is a simplified example. The flowchart for an actual processor would be more complex. In any case, we have reached the point in our discussion in which the operation of the processor is defined as the performance of a sequence of micro operations. We can now consider how the control unit causes this sequence to occur.



3.2 CONTROL OF THE PROCESSOR

As a result of our analysis in the preceding section, we have decomposed the behavior or functioning of the, processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen. Thus, we can define the *functional requirements* for the control unit: those functions that the control unit must perform. A definition of these functional requirements is the basis for the design and implementation of the control unit.

With the information at hand, the following three-step process leads to a characterization of the control unit:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

We have already performed steps 1 and 2. Let us summarize the results. First, the basic functional elements of the processor are the following:

- ALU
- Registers
- Internal data paths External data paths
- Control unit

Some thought should convince you that this is i complete list. The ALU is the functional essence of the computer. Registers are used to store data internal to the processor. Some registers contain status information needed to manage instruction sequencing (e.g., a program status word). Others contain data that go to or come from the ALU, memory, and I/O modules. Internal data paths are used to move data between registers and between register and ALU. External data paths link registers to memory and 1/O modules, often by means of a system bus. The control unit causes operations to happen within the processor.

The execution of a program consists of operations involving these processor elements. As we have seen, these operations consist of a sequence of micro-operations. micro-operations fall into one of the following categories:

- Transfer data from one register to another.
- Transfer data from a register to an external interface (e.g., system bus).
- Transfer data from an external interface to a register.
- Perform an arithmetic or logic operation, using registers for input and output.

All of the micro-operations needed to perform one instruction cycle, including all of the micro-operations to execute every instruction in the instruction set, fall into one of these categories.

We can now be somewhat more explicit about the way in which the control unit functions. The control unit performs two basic tasks:

- **Sequencing:** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
- **Execution:** The control unit causes each micro-operation to be performed.

The preceding is a functional description of what the control unit does. The key to how the control unit operates is the use of control signals.

Controls Signals

We have defined the elements that make up the processor (ALU, registers, data paths) and the micro-operations that are performed. For the control unit to perform its function, it must have inputs that allow it to determine the state of the system and outputs that allow it to control the behavior of the system. These are the external specifications of the control unit. Internally, the control unit must have the logic required to perform its sequencing and execution functions. The remainder of this section is concerned with the interaction between the control unit and the other elements of the processor.

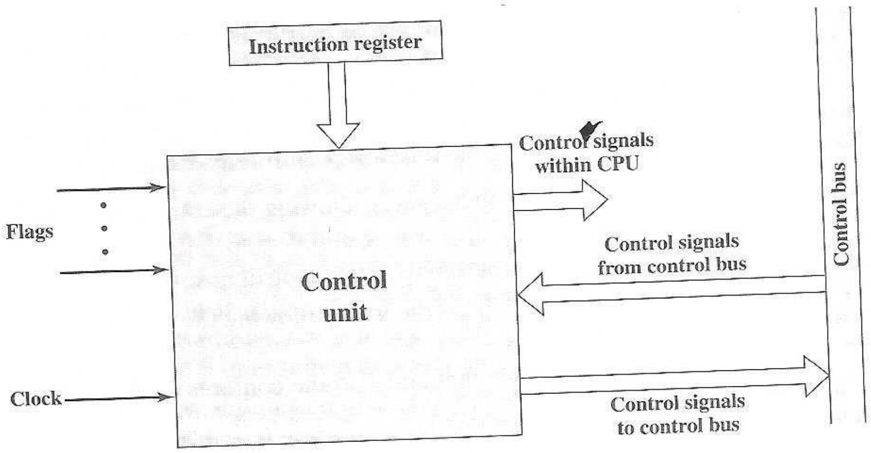

Instruction register
Control signals within CPU
Flags
Control unit
Control signals from control bus
Control bus
Clock
Control signals to control bus

Figure 3.2.1 is a general model of the control unit, showing all of its inputs and outputs. The inputs are

- ✓ **Clock:** This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time, or the clock cycle time.
- ✓ **Instruction register:** The opcode and addressing mode of the current instruction are used to determine which micro-operations to perform during the execute cycle.
- ✓ **Flags**: These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example, for the increment-and-skip-if-zero (ISZ) instruction, the control unit will increment the PC if the zero flag is set.

Control signals from control bus: The control bus portion of the system bus provides signals to the control unit.

The outputs are as follows:

- ✓ Control signals within the processor: These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.
- Control signals to control bus: These are also of two types: control signals to memory, and control signals to the I/O modules.

Three types of control signals are used: those that activate an ALU function, those that activate a data path, and those that are signals on the external system bus or other external interface. All of these signals are ultimately applied directly as binary inputs to individual logic gates.

Let us consider again the fetch cycle to see how the control unit maintains control. The control unit keeps track of where it is in the instruction cycle. At a given point, it knows that the fetch cycle is to be performed next. The first step is to transfer the contents of the PC to the MAR. The control unit does this by activating the control signal that opens the gates between the bits of the PC and the bits of the MAR. The next step is to read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:

A control signal that opens gates, allowing the contents of the MAR onto the address bus

A memory read control signal on the control bus

A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR

Control signals to logic that add 1 to the contents of the PC and store the result back to the PC

Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

This completes the fetch cycle except for one thing: The control unit must decide whether to perform an indirect cycle or an execute cycle next. To decide this, it examines the IR to see if an indirect memory reference is made.

The indirect and interrupt cycles work similarly. For the execute cycle, the control unit begins by examining the opcode and, on the basis of that, decides which sequence of micro-operations to perform for the execute cycle.

To illustrate the functioning of the control unit, let us examine a simple example. Figure 3.2.3 illustrates the example. This is a simple processor with a single accumulator

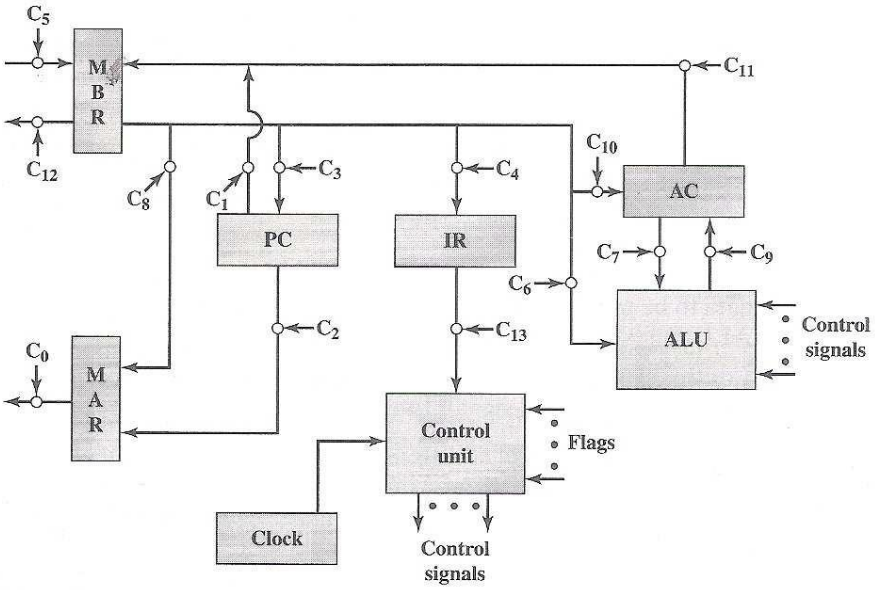


Figure 15.5 Data Paths and Control Signals

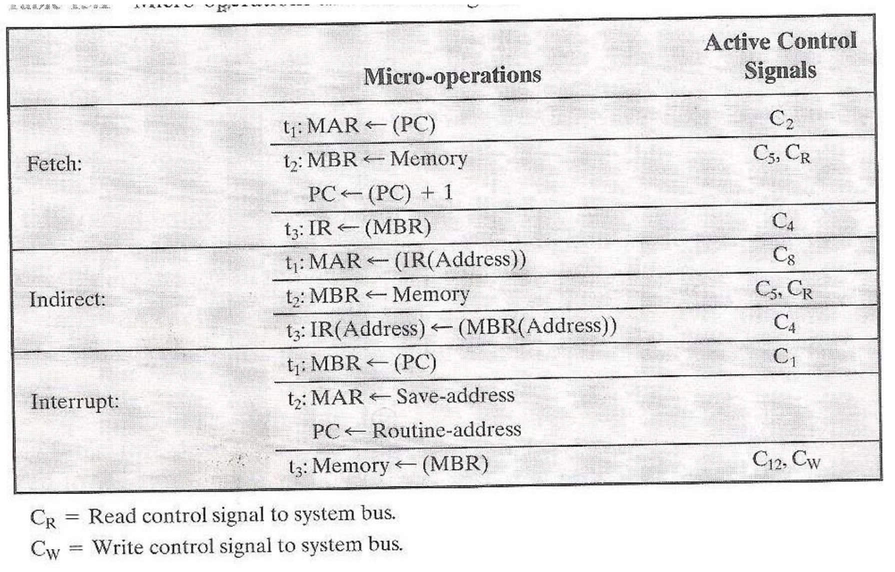
(AC). The data paths between elements are indicated. The control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled C_i and indicated by a circle. The control unit receives inputs from the clock, the instruction register, and flags. With each **dock** cycle, the control unit reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

Data paths: The control unit controls the internal flow of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the instruction register. For each path to be controlled, there is a switch (indicated by a circle in the figure). A control signal from the control unit temporarily opens the gate to let data pass.

ALU: The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic circuits and gates within the ALU.

System bus: The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.



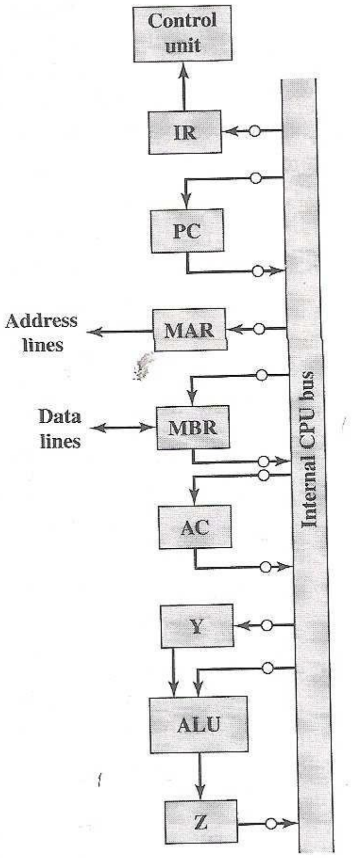
	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

C_R = Read control signal to system bus.
C_W = Write control signal to system bus.

It is worth pondering the minimal nature of the control unit. The control unit is the engine that runs the entire computer. It does this based only on knowing the instructions to be executed and the nature of the results of arithmetic and logical operations (e.g., positive, overflow, etc.). It never gets to spe the data being processed or the actual results produced. And it controls everything with a few control signals to points within the processor and a few control signals to the system bus.

Figure 15.5 indicates the use of a variety of data paths. The complexity of this type of organization should be clear.

Using an internal processor bus, Figure 3.2.2 can be rearranged as shown in Figure 3.2.4 A single internal bus connects the ALU and all processor registers.



Gates and control signals are provided for movement of data onto and off the bus from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

Two new registers, labeled Y and Z, have been added to the organization. These are needed for the proper operation of the ALU. When an operation involving two operands is performed, one can be obtained from the internal bus, but the other must be obtained from another source. The AC could be used for this purpose, but this limits the flexibility of the system and would not work with a processor with multiple general-purpose registers. Register Y provides temporary storage for the other input. The ALU is a combinatorial circuit with no internal storage. Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Thus, the output of the ALU cannot be directly connected to the bus, because this output would feed back to the input. Register Z provides temporary output storage. With this arrangement, an operation to add a value from memory to the AC would have the following steps:

t_1:	MAR	←	(IR (address))
t_2:	MBR	←	Memory
t_3:	Y	←	(MBR)
t_4:	Z	←	(AC) + (Y)
t_5:	AC	←	(Z)

Other organizations are possible, but, in general, some sort of internal bus or set of internal buses is used. The use of common data paths simplifies the interconnection layout and the control of the processor. Another practical reason for the use of an internal bus is to save space.

To illustrate some of the concepts introduced thus far in this unit, let us consider the Intel 8085. Its organization is shown in Figure 3.2.5. Several key components that may not be self-explanatory are:

- **Incremental decrementer address latch:** Logic that can add 1 to or subtract 1 from the contents of the stack pointer or program counter. This saves time by avoiding the use of the ALU for this purpose.
- **Interrupt control:** This module handles multiple levels of interrupt signals.
- **Serial UO control:** This module interfaces to devices that communicate 1 bit at a time.

Table 15.2 describes the external signals into and out of the 8085. These are linked to the external system bus. These signals are the interface between the 8085 processor and the rest of the system (Figure 15.8).

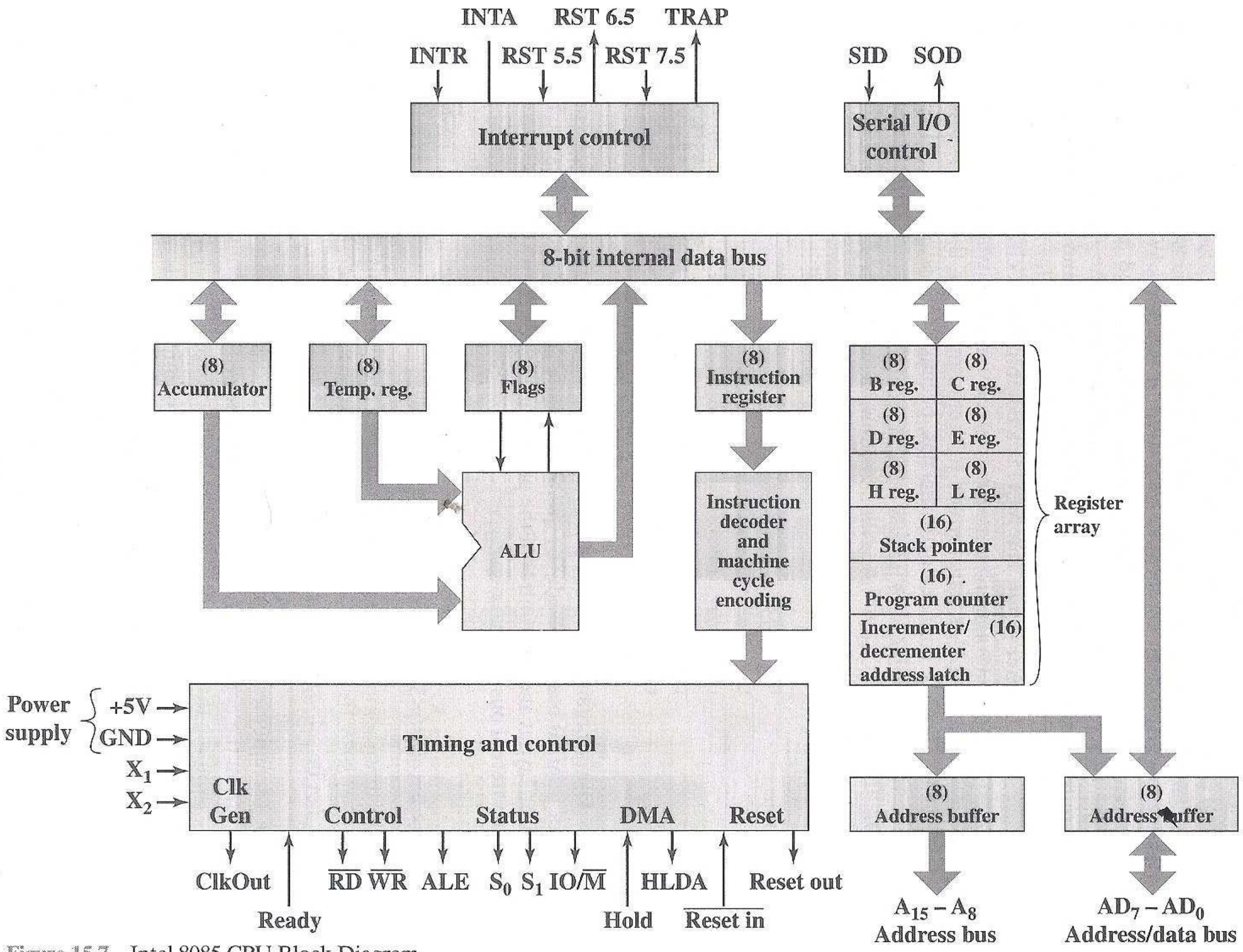
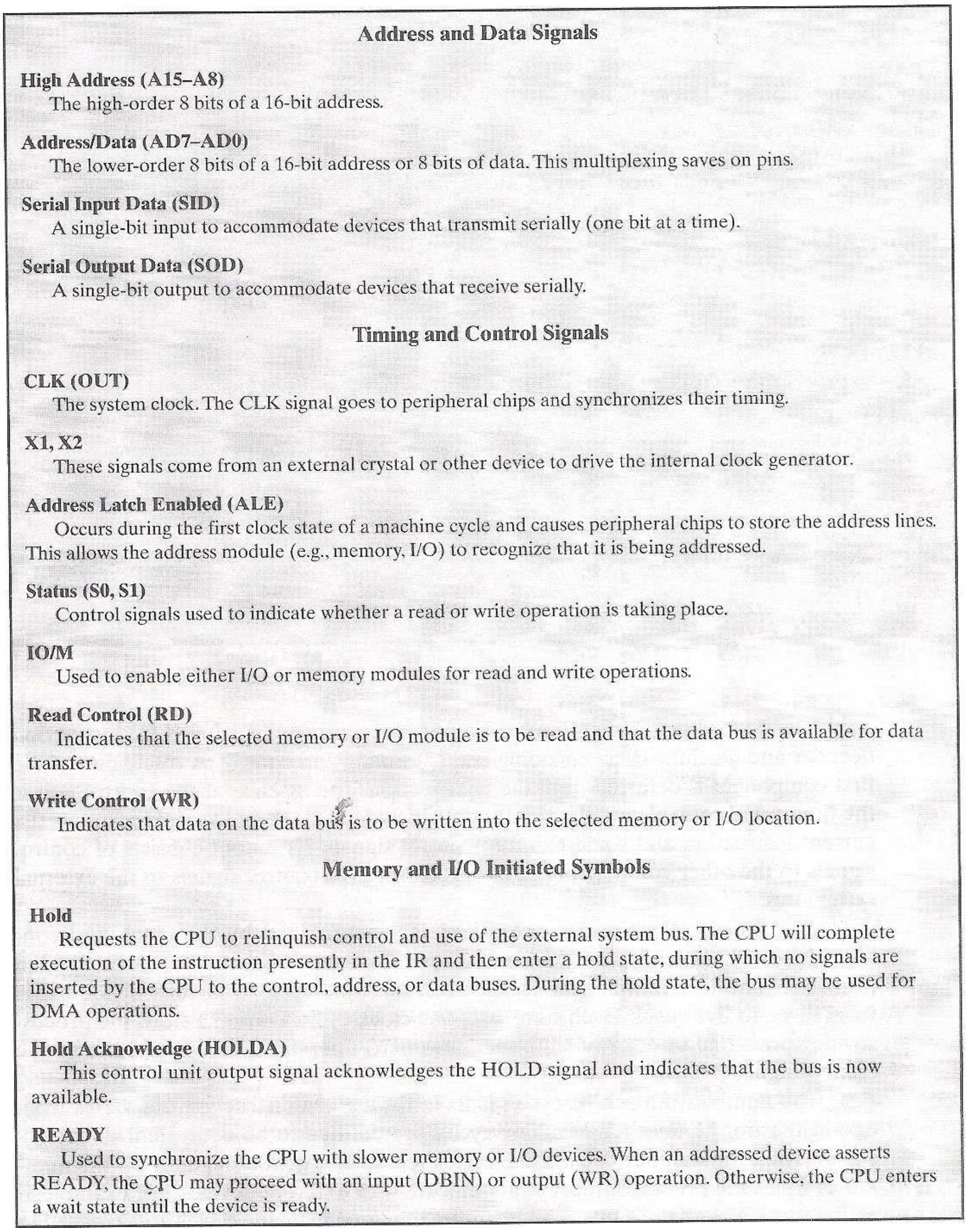


Figure 15.7 Intel 8085 CPU Block Diagram



Address and Data Signals

High Address (A15–A8)
The high-order 8 bits of a 16-bit address.

Address/Data (AD7–AD0)
The lower-order 8 bits of a 16-bit address or 8 bits of data. This multiplexing saves on pins.

Serial Input Data (SID)
A single-bit input to accommodate devices that transmit serially (one bit at a time).

Serial Output Data (SOD)
A single-bit output to accommodate devices that receive serially.

Timing and Control Signals

CLK (OUT)
The system clock. The CLK signal goes to peripheral chips and synchronizes their timing.

X1, X2
These signals come from an external crystal or other device to drive the internal clock generator.

Address Latch Enabled (ALE)
Occurs during the first clock state of a machine cycle and causes peripheral chips to store the address lines. This allows the address module (e.g., memory, I/O) to recognize that it is being addressed.

Status (S0, S1)
Control signals used to indicate whether a read or write operation is taking place.

IO/M
Used to enable either I/O or memory modules for read and write operations.

Read Control (RD)
Indicates that the selected memory or I/O module is to be read and that the data bus is available for data transfer.

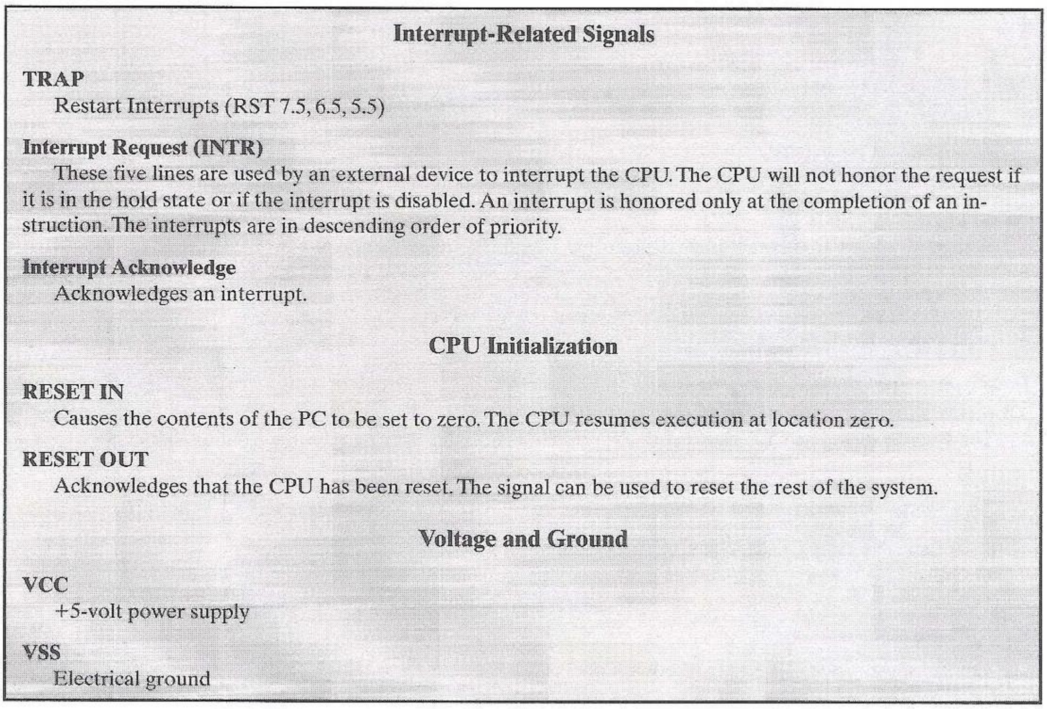
Write Control (WR)
Indicates that data on the data bus is to be written into the selected memory or I/O location.

Memory and I/O Initiated Symbols

Hold
Requests the CPU to relinquish control and use of the external system bus. The CPU will complete execution of the instruction presently in the IR and then enter a hold state, during which no signals are inserted by the CPU to the control, address, or data buses. During the hold state, the bus may be used for DMA operations.

Hold Acknowledge (HOLDA)
This control unit output signal acknowledges the HOLD signal and indicates that the bus is now available.

READY
Used to synchronize the CPU with slower memory or I/O devices. When an addressed device asserts READY, the CPU may proceed with an input (DBIN) or output (WR) operation. Otherwise, the CPU enters a wait state until the device is ready.



Interrupt-Related Signals

TRAP
Restart Interrupts (RST 7.5, 6.5, 5.5)

Interrupt Request (INTR)
These five lines are used by an external device to interrupt the CPU. The CPU will not honor the request if it is in the hold state or if the interrupt is disabled. An interrupt is honored only at the completion of an instruction. The interrupts are in descending order of priority.

Interrupt Acknowledge
Acknowledges an interrupt.

CPU Initialization

RESET IN
Causes the contents of the PC to be set to zero. The CPU resumes execution at location zero.

RESET OUT
Acknowledges that the CPU has been reset. The signal can be used to reset the rest of the system.

Voltage and Ground

VCC
+5-volt power supply

VSS
Electrical ground

The control unit is identified as having two components labeled (1) instruction decoder and machine cycle encoding and (2) timing and control. A discussion of the first component is deferred until the next section. The essence of the control unit is the timing and control module. This module includes a clock and accepts as inputs the current instruction and some external control signals. Its output consists of control signals to the other components of the processor plus control signals to the external system bus.

The timing of processor operations is synchronized by the clock and controlled by the control unit with control signals. Each instruction cycle is divided into from one to five *machine* cycles; each machine cycle is in turn divided into from three to five *states.* Each state lasts one clock cycle. During a state, the processor performs one or a set of simultaneous micro-operations as determined by the control signals.

The number of machine cycles is fixed for a given instruction but varies from one instruction to accesses. Thus, the number of machine cycles for an instruction depends on t¯ lie number of times the processor must communicate with external devices. For example, if an instruction consists of two 8-bit portions, then two machine cycles are required to fetch the instruction. If that instruction involves a 1-byte memory or I/O operation, then a third machine cycle is required for execution.

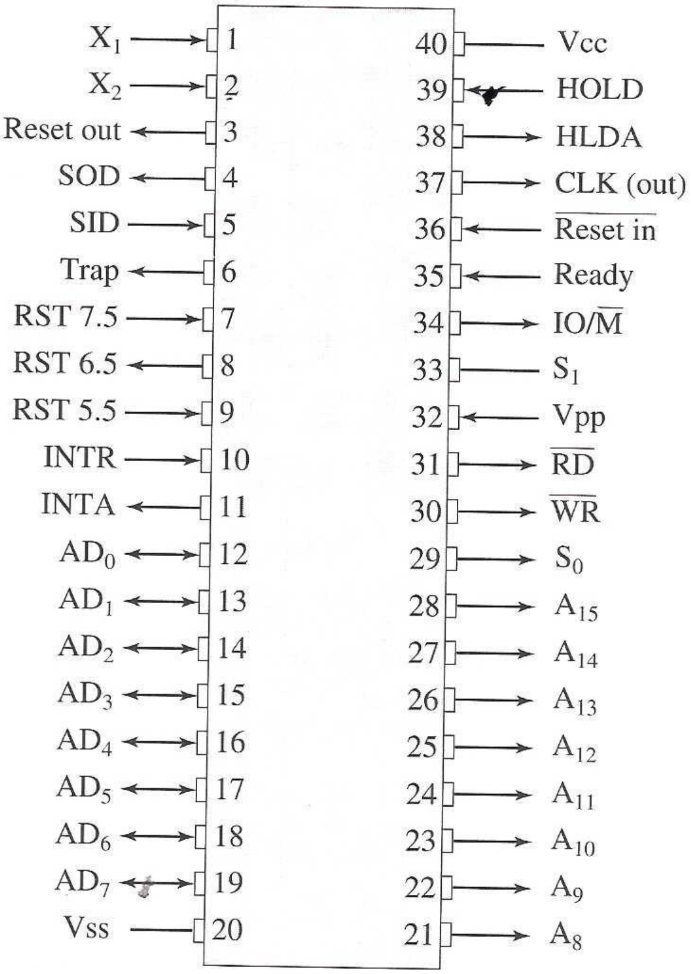


Figure 3.2.7 gives an example of 8085 timing, showing the value of external control signals. Of course, at the same time, the control unit generates internal control signals that control internal data transfers. The diagram shows the instruction cycle for an OUT instruction. Three machine cycles (M_1, M_Z, M_3) are needed. During the first, the OUT instruction is fetched. The second machine cycle fetches the second half of the instruction, which contains the number of the I/O device selected for output. During the third cycle, the contents of the AC are written out to the selected device over the data bus.

pulse signals the start of each machine cycle from the control unit. The ALE pulse alerts external circuits. During timing state T_1 of machine cycle M_l, the control unit sets the IO/M signal to indicate that this is a memory operation. Also, the control unit causes the contents of the PC to be placed on the

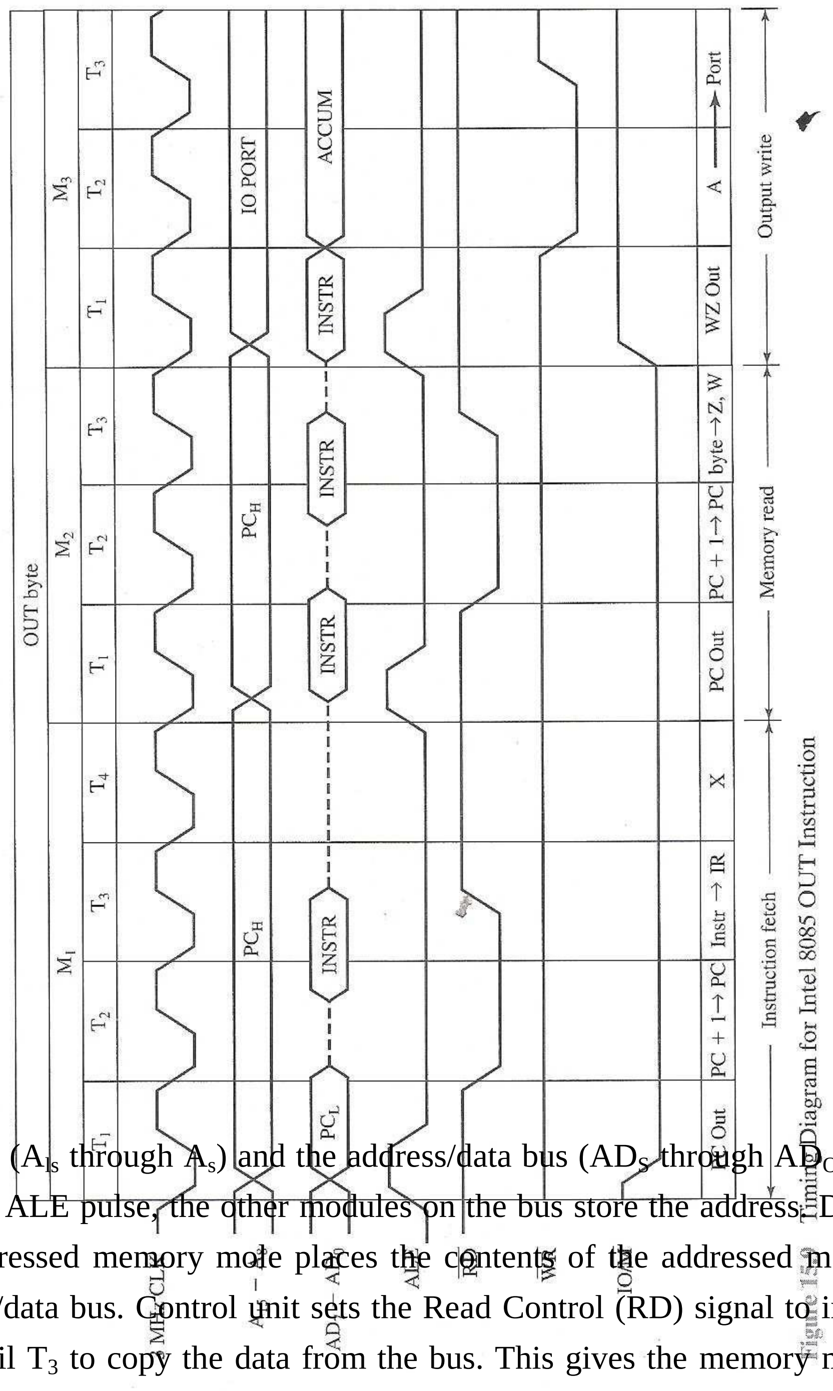


Figure 15.9 Timing Diagram for Intel 8085 OUT Instruction

address bus (A_{1s} through A_s) and the address/data bus (AD_S through AD_O). With the falling edge of the ALE pulse, the other modules on the bus store the address. During timing state T_2, the addressed memory mode places the contents of the addressed memory location on the address/data bus. Control unit sets the Read Control (RD) signal to indicate a read, but it waits until T_3 to copy the data from the bus. This gives the memory module time to put the data on the bus and for the signal levels to stabilize. The final state, T_4, is a bus *idle* state during which the processor decodes the instruction. The remaining machine cycles proceed in a similar fashion.

3.3 HARDWIRED CONTROL/ IMPLEMENTATION

In a hardwired implementation, the control unit is essentially a state machine circuit. Its input logic signals are transformed into a set of output logic signals, which are the control signals.

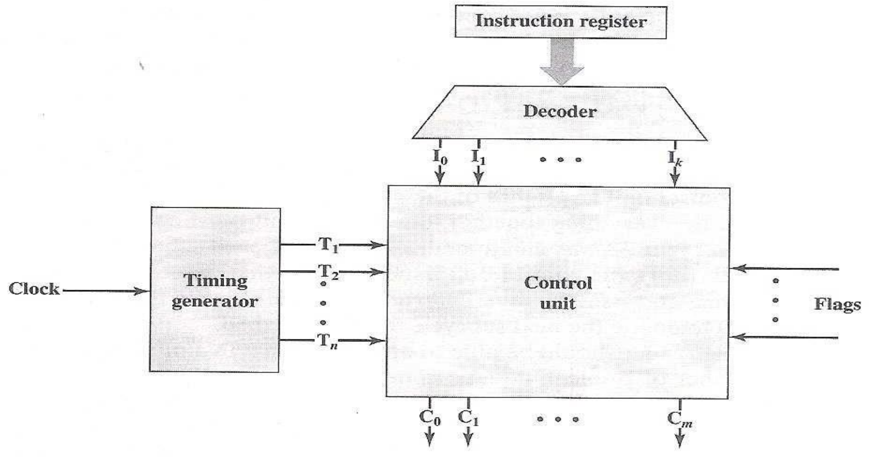
3.3.1 CONTROL UNIT INPUT

The key inputs are the instruction registers, the clock, flags and control bus signals. In the case of the flags and control bus signals, each individual bit typically has some meaning (eg overflow). The other two inputs, however are not directly useful to the control unit. First consider the instruction register. The control unit makes use of the opcode and will perform different actions (issue a different combination of control signals) for different instructions. To simplify the control unit logic, there should be a unique logic input for each opcode. This function can be performed by a decoder, which takes an encoded input and produces a single output.

The clock portion of the control unit issues a representative sequence of pulses. This is useful for measuring the duration of micro-operations. Essentially the period of the clock pulses must be long enough to allow the propagation of signals along data paths and through processor circuitry. However the control unit emits different control signals at different time units within a single instruction cycle. Thus, we would like a counter as input to the control unit with a different control signal being used for T1, T2 and so forth. At the end of an instruction cycle, the control unit must feed back to the counter to reinitialize it at T1.

Table 15.3 A Decoder with Four Inputs and Sixteen Outputs

I1	I2	I3	I4	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13	O14	O15	O16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



With these two refinements, the control unit can be depicted as in Figure 15.10.

To define the hardwired implementation of a control unit, all that remains is to discuss the internal logic of the control unit that produces output control signals as a function of its input signals.

Essentially, what must be done is, for each control signal, to derive a Boolean expression of that signal as a function of the inputs. This is best explained by example. Let us consider again our simple example illustrated in Figure 15.5. We saw in Table 15.1 the micro-operation sequences and control signals needed to control three of the four phases of the instruction cycle.

Let us consider a single control signal, C_5. This signal causes data to be read from the external data bus into the MBR. We can see that it is used twice in Table 15.1. Let us define two new control signals, P and Q, that have the following interpretation:

PQ = 00	Fetch Cycle
PQ = Ol	Indirect Cycle
PQ = 10	Execute Cycle
PQ = 11	Interrupt Cycle

Then the following Boolean expression defines C5:

$$C_5 = P^{\cdot}Q^{\cdot}T_2 + P.Q.T2$$

That is, the control signal C5 will be asserted during the second time unit of both the fetch and indirect cycles.

This expression is not complete. C_5 is also needed during the execute cycle. For our simple example, let us assume that there are only three instructions that read from memory: LDA, ADD, and AND. Now we can define C_5 as

C_5 + P . Q . T_Z + P - Q - (LDA + ADD + AND)-T2

This same process could be repeated for every control signal generated by the processor. The result would be a set of Boolean equations that define the behavior of the control unit and hence of the processor.

To tie everything together, the control unit must control the state of the instruction cycle. As was mentioned, at the end of each sub cycle (fetch, indirect, execute, interrupt), the control unit issues a signal that causes the timing generator to reinitialize and issue T_1. The control unit must also set the appropriate values of P and Q to define the next sub cycle to be performed.

The reader should be able to appreciate that in a modern complex processor, the number of Boolean equations needed to define the control unit is very large. The task of implementing a combinatorial circuit that satisfies all of these equations becomes extremely difficult. The result is that a far simpler approach, known as microprogramming, is usually used.

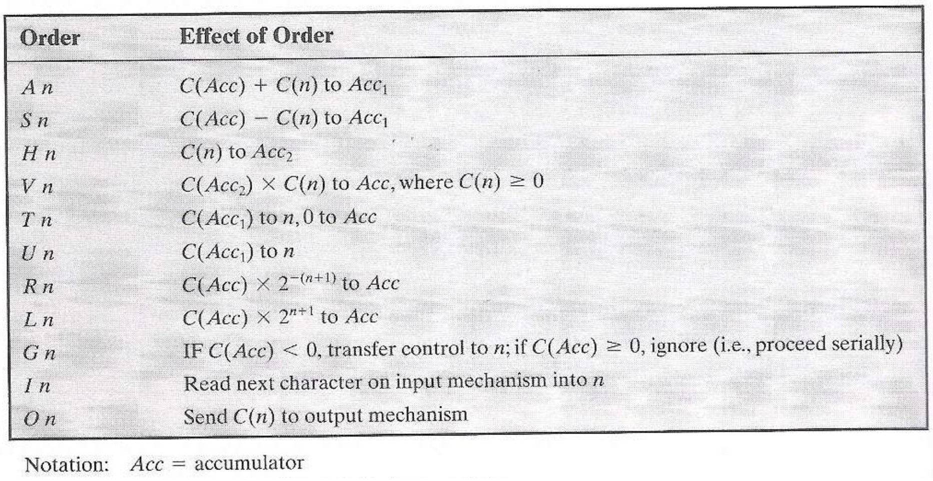
3.4 MICRO PROGRAMMED CONTROL

An alternative to a hardwired control unit is a micro programmed control unit in which the logic of the control unit is specified by a micro program. A micro program consist of a sequence of instructions in a microprogramming language. These are very simple instruction that specify micro operations.

3.4.1 MICRO INSTRUCTIONS

To implement a control unit as n interconnection of basic logic elements is no easy task. The design must include logic for sequencing through micro-operation for executing micro- operations, for interpreting opcodes and for making decisions based in ALU flags.

Table 16.1 Machine Instruction Set for Wilkes Example



Order	Effect of Order
A n	$C(Acc) + C(n)$ to Acc_1
S n	$C(Acc) - C(n)$ to Acc_1
H n	$C(n)$ to Acc_2
V n	$C(Acc_2) \times C(n)$ to Acc, where $C(n) \geq 0$
T n	$C(Acc_1)$ to n, 0 to Acc
U n	$C(Acc_1)$ to n
R n	$C(Acc) \times 2^{-(n+1)}$ to Acc
L n	$C(Acc) \times 2^{n+1}$ to Acc
G n	IF $C(Acc) < 0$, transfer control to n; if $C(Acc) \geq 0$, ignore (i.e., proceed serially)
I n	Read next character on input mechanism into n
O n	Send $C(n)$ to output mechanism

Notation: Acc = accumulator

relatively inflexible. For example, it is difficult to change the design if one wishes tc add a new machine instruction.

An alternative, which has been used in many CISC processors, is to implement a microprogrammed control unit.

Consider Table 16.1. In addition to the use of control signals, each micro-operation is described in symbolic notation. This notation looks suspiciously like a programming language. In fact it is a language, known as a **microprogramming language**. Each line describes a set of micro-operations occurring at one time and is known as a **microinstruction**. A sequence of instructions is known as a **microprogram,** or ***firmware.*** This latter term reflects the fact that a microprogram is midway between: hardware and software. It is easier to design in firmware than **hardware**, but it is more difficult to write a firmware program than a software program.

How can we use the concept of microprogramming to implement a contra: unit? Consider that for each micro-operation, all that the control unit is allowed t o do is generate a set of control signals. Thus, for any micro-operation, each control link: emanating from the control unit is either on or off. This condition can, of course, be represented by a binary digit for each control line. So we could construct a contra *word* in which each bit represents one control line. Then each micro-operation would be represented by a different pattern of 1s and Os in the control word.

Suppose we string together a sequence of control words to represent the sequence of micro-operations performed by the control unit. Next, we must recognizz that the sequence of micro-operations is not fixed. Sometimes we have an indireccycle; sometimes we do not. So let us put our control words in a memory, with each word having a unique address. Now add an address field to each control word,

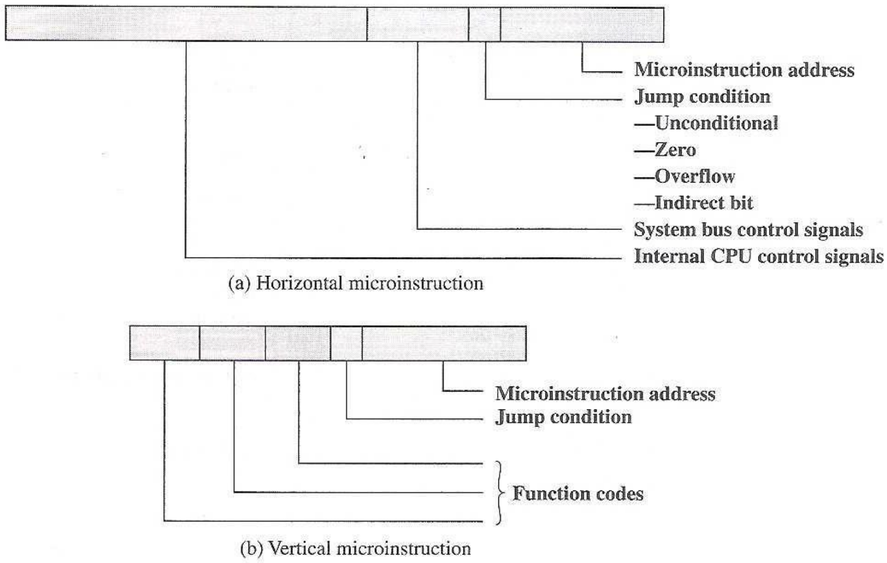


Figure 16.1 Typical Microinstruction Formats

indicating the location of the next control word to be executed if a certain condition is true (e.g., the indirect bit in a memory-reference instruction is 1). Also, add a few bits to specify the condition.

The result is known as a horizontal microinstruction, an example of which is shown in Figure 3.4.1a The format of the microinstruction or control word is as follows. There is one bit for each internal processor control line and one bit for each system bus control line There is a condition field indicating the condition under which there should be a' branch, and there is a field with the address of the microinstruction to be executed next when a branch is taken. Such a microinstruction is interpreted as follows:

- To execute this microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed.
- If the condition indicated by the condition bits is false, execute the next microinstruction in sequence.
- If the condition indicated by the condition bits is true, the next microinstruction to be executed is indicated in the address field.

Figure 3.4.1b shows how these control words or microinstructions could be arranged in a control memory. The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating where to go next. There is a special execute cycle routine whose only purpose is to signify that one of the machine instruction routines (AND, ADD, and so on) is to be executed next, depending on the current opcode.

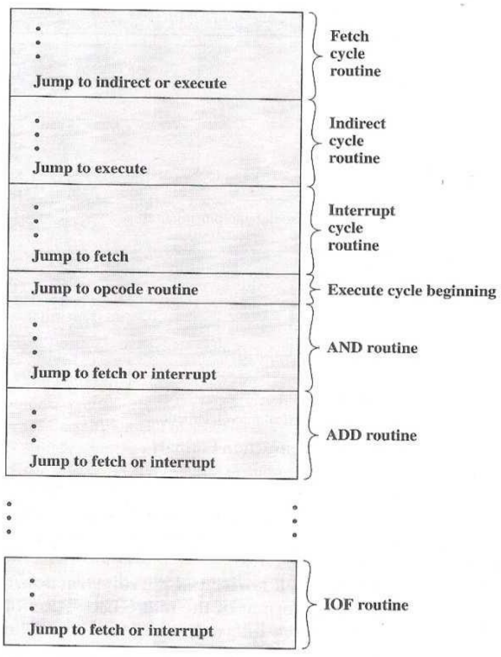
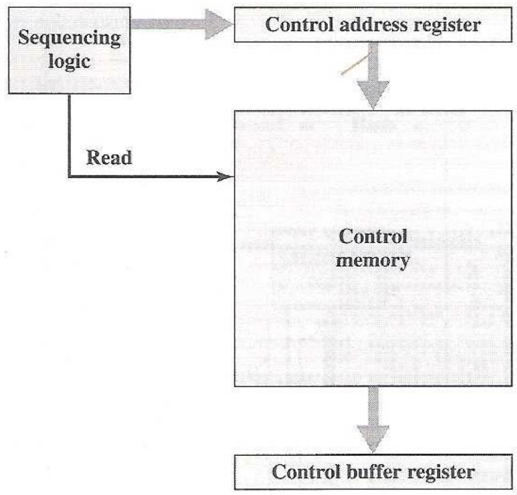


Figure 16.2 Organization of Control Memory

The control memory of Figure 16.2 is a concise description of the complete operation of the control unit. It defines the sequence of micro-operations to be performed during each cycle (fetch, indirect, execute, interrupt), and it specifies the sequencing of these cycles. If nothing else, this notation would be a useful device for documenting the functioning of a control unit for a particular computer. But it is more than that. It is also a way of implementing the control unit.

The control memory of Figure 3.4.1b contains a program that describes the behavior of the control unit. It follows that we could implement the control unit by simply executing that program.

Figure 3.4.1b shows the key elements of such an implementation. The set of microinstructions is stored in the *control memory*. The *control address register* contains the address of the next microinstruction to be read. When a microinstruction is read from the control memory, it is transferred to a control buffer register the left-hand portion of that register (see Figure 3.4.1b) connects to the control lines emanating



from the control unit. Thus, *reading* a microinstruction from the control memory is the same as *executing* that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.

Let us examine this structure in greater detail, as depicted in Figure 3.4.1d Comparing this with Figure 3.4.1d we see that the control unit still has the same inputs (IR, ALU flags, clock) and outputs (control signals). The control unit functions as follows:

1. To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
2. The word whose address is _,Specified in the control address register is read into the control buffer register.
3. The content of the control buffer register generates control signals and next address information for the sequencing logic unit.
4. The sequencing logic unit loads a new address into the control address register based on the next-address information from the control buffer register and the ALU flags.

All this happens during one clock pulse.

The last step just listed needs elaboration. At the conclusion of each microinstruction, the sequencing logic unit loads a new address into the control address register. Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

- Get the next instruction: Add 1 to the control address register.

- Jump to a new routine based on a jump microinstruction: Load the address field of the control buffer register into the control address register.
- Jump to a machine instruction routine: Load the control address register based on the opcode in the IR.

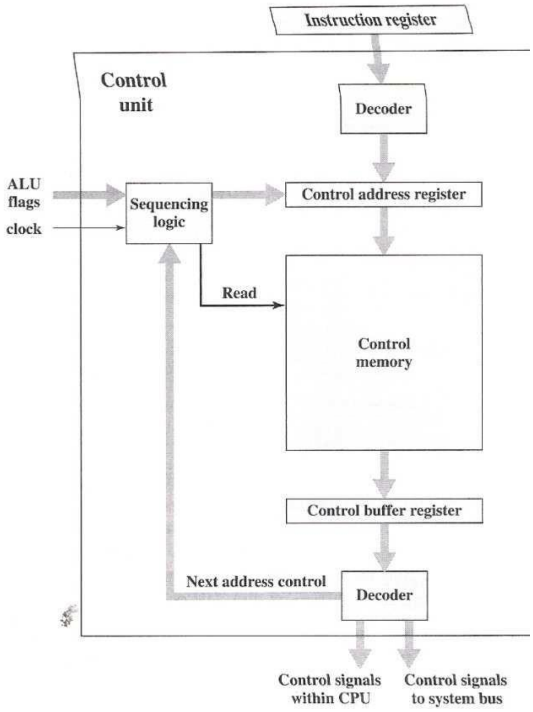


Figure 3.4.1d shows two modules labeled *decoder.* The upper decoder translates the opcode of the IR into a control memory address. The lower decoder is not used for horizontal microinstructions but is used for vertical microinstructions (Figure 16.1b). As was mentioned, in a horizontal microinstruction every bit in the control' field attaches to a control line. In a vertical microinstruction, a code is used for each action to be performed [e.g., MAR F- (PC)], and the decoder translates this code into individual control signals. The advantage of vertical microinstructions is that they are more compact (fewer bits) than horizontal microinstructions, at the expense of a small additional amount of logic and time delay.

3.4.2 ADVANTAGES AND DISADVANTAGES

The principal advantage of the use of micro-programming to implement a control unit is that it simplifies the design of the control unit. Thus it is both cheaper and less error prone to implement. A hard wired control unit must contain complex logic for sequencing through the many micro-operation s of the instructions cycle. On the other hand the decoders and sequencing logic unit of a micro programmed control unit are very simple pieces of logic.

The principal disadvantage of a micro programmed unit is that it will be somewhat slower than a hardwired unit of comparable technology. Despite this, microprogramming is the dominant technique for implementing control units in pure CISC architecture due to its ease of implementation. RISC processors with their simpler instruction format, typically use hardwired control units

The two basic task performed by a micro programmed control unit arer as follows:

- Micro instruction sequencing: Get the next control signals needed to execute the micro instruction. In designing a control unit, these tasks must be considered together because both affect the format of the micro instruction and the timing of the control unit.

4.0 CONCLUSION

Micro- operations are the functional or atomic operations of a processor. The concepts of micro- operation serve as a guide to the design of the control unit.

5.0 SUMMARY

In each instruction cycle is made up of a set of micro-operations that generates control signals. Execution is accomplished by the effect of these control signals, emanating for the control unit to the ALU registers and system interconnection structure. Finally an approach to the implements of the control unit referred to as hard wired implementation is presented. Furthermore, the concept of micro- operations leads to an elegant and powerful approach to control unit implementation, known as micro programming. Besides each instruction in the machine language of the processor is translated into a sequence of lower- level control unit instruction referred to as micro instructions and the process of translation is referred to as micro programming.

6.0 TUTOR- MARKED ASSIGNMENT

1. What is the relationship between instructions and micro operations?
2. Briefly what is meant by a hard wired implementation of a control unit.
3. What are the basic tasks performed by a micro programmed control unit?
4. What is the difference between a hard wired implementation and a micro programmed implementation of a control unit?

7.0 REFERENCES/FURTHER READING

Carter J. Microprocessor Architecture and Microprogramming – Upper saddle River N. J Prentice HALL, 1996

Module 3 PARALLEL ORGANIZATION

UNIT 1: Multiple processor organization
UNIT 2: Symmetric Multiprocessor
UNIT 3: Multithreading and chip multi processors
UNIT 4: Vector computation

Unit 1: Multiple processor organization

1.0 INTRODUCTION

At the micro- operation level multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel. This approach is taken further with super scalar organization, which exploits instruction- level parallelism. With a super scalar machine, there are multiple instructions for the same program in parallel. As computer technology has evolved and as the cost of computer hard ware has dropped computer designers have sought more and more opportunities for parallelism usually to enhance performance and in some cases to increase availability.

2.0 OBJECTIVES

At the end of the unit you should be able to

- Understand the traditional way to increase system performance.
- Explain and discuss symmetric multi processor (SMPs) and clusters
- Explain and discuss chip multiprocessing and multi threaded processor.

3.1 TYPES OF PARALLEL PROCESSOR SYSTEM

A taxonomy first introduced by Flynn is still the most common way of categorizing system with parallel processing capability. He proposed the following categories of computer systems:

- **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.
- **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lock step basis. Each processing element has associated data memory so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category
- **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.
- **Multiple instructions, multiple data (MIMD) stream:** A sequence of data transmitted to a set of processors, each of which executes a different instruction sequence. This structure is none commercially implemented.
- **Multiple instructions, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters and NUMA systems fit into this category.
- With the MMD organization, the processors are general-purpose; each is able to process all of the instruction necessary to perform the appropriate data transformation. MIMDs can be further sub divided by the means in which the processors communicate (Figure 3. 1. 1) if the processors have a common memory, then each processor accesses programs and data stores in the shared memory and processors.

4.0 CONCLUSION

A traditional way to incrust system performance is to use multiple processors that can execute in parallel to support a given work load. The two most common multiple processor organizations are symmetric multi processors (SMPs) and clusters. More recently, non uniform memory access (NUMA) systems have been introduced commercially.

5.0 SUMMARY

In a parallel organization, multiple processing units cooperate to execute applications whereas a superscalar process exploits opportunities for parallel execution at the instruction level, a parallels processing organization looks for a grosser level of parallelism one that enables work to be done in parallel and cooperatively by multiple processors

6.0 TUTOR- MARKED ASSIGNMENT

1. List and briefly define types of parallel processor system.
2. List the two most common multiple processor organizations

7.0 REFERENCES/FURTHER READING

Catanzaro B. Multi processor system Architecture Mountain View CA, Sun sift pres 1994

UNIT 2: SYMMETRIC MULTI PROCESSOR

1.0 Introduction

Virtually all single user personal computers and most work stations contained a single generate purpose micro processors. As demand for performance increases and is the cost of microprocessors continues to drop. Vendors have introduced system with and SMP organization.

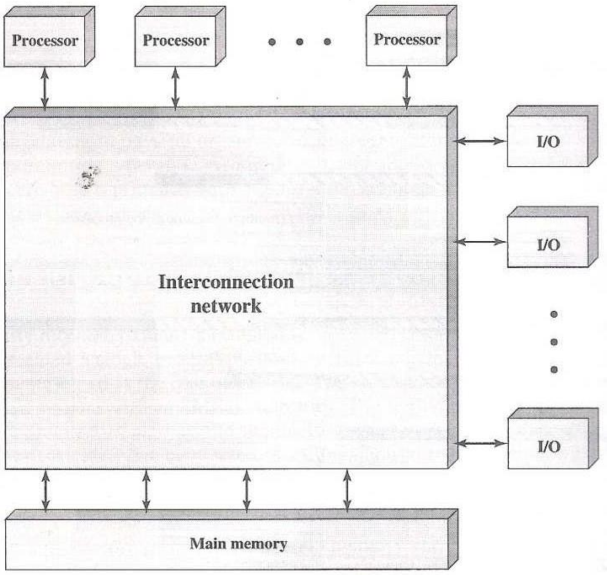
2.0 **Objectives**

At the end of this unit you should be able to

- Understand the organization of a multi processor system
- Explain the character of an SMP as a standalone computer system.

3.1 Organization

This depicts in general terms the organization of a multiprocessor system. There are two or more processors. Each processor is self-contained, including a control unit, ALU, registers, and, typically, one or more levels of cache. Each processor



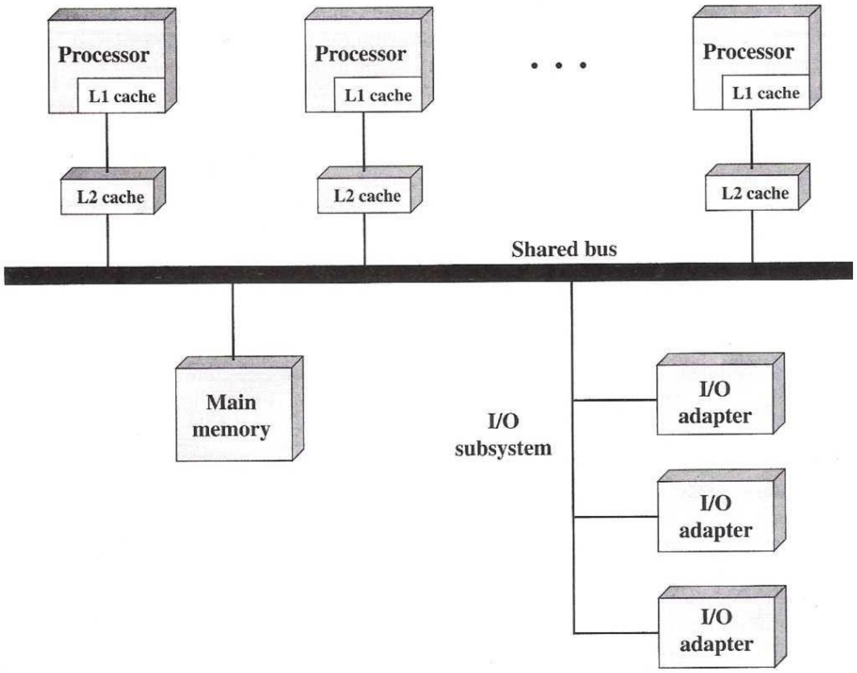


Figure 17.5 Symmetric Multiprocessor Organization

has access to a shared in pin memory and the I/O devices through some form of interconnection mechanism. The processors can communicate with each other through memory (messages and status information left in common data areas). It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible. In some configurations, each processor may also have its own private main memory and I/O channels in addition to the shared resources.

The most common organization for personal computers, workstations, and servers is the time-shared bus. The time-shared bus is the simplest mechanism for constructing a multiprocessor system (Figure 3.1.2). The structure and interfaces are basically the same as for a single-processor system that uses a bus interconnection. The bus consists of control, address, and data lines. To facilitate DMA transfers from I/O processors, the following features are provided:

- **Addressing:** It must be possible to distinguish modules on the bus to determine the source and destination of data.
- **Arbitration:** Any I/Q module can temporarily function as "master" A mechanism is provided to arbitrate competing requests for bus control, using some sort of priority scheme.
- **Time – sharing:** When one module is controlling the bus other modules are locked out and must if necessary, suspend operation until bus access is achieved. These uniprocessor features are directly usable in an SMP organization. In this latter case, there are now multiple processors as well as multiple I/O processors all attempting to gain access to one or more memory modules via the bus. The bus organization has several attractive features:
- **Simplicity**: This is the simplest approach to multi processor organization. The physical feature interface and the addressing, arbitration and time sharing logic of each processor remain the same as in a single processor system.
- **Flexibility:** It is generally easy to expand the system by attracting more processors to the bus.
- **Reliability:** The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system.

The main draw back to the bus organization is performance. All memory references pass through the common bus. Thus the bus cycle time limits the speed of the system. To improve performance, it is desirable to equip each processor with a cache memory. This should reduce the number of bus a access dramatically. Typically work station and PC SMPs have two levels of cache, with the LI cache internal (same chip as the processor) and the L2 cache either internal or external some processors now employ a L3 cache as well. The use of cache contains introduces some new designs considerations. Because each local cache contains an image of a portion of memory if a word is altered in one cache, it could conceivably invalidate a word in another cache. To prevent this other processessor must be altered that an update has taken place. This problem is known as the cache coherence problem and is typically addressed in hardware rather than by the operating system.

3.2 MULTIPROCESSOR OPERATING DESIGN CONSIDERATIONS

An SMP operating system manages processor and other computer resources so that the user perceives a single operating system controlling system controlling system resources. In fact such a configuration should appear as a single processor multi programming system. In both the SMP and uniprocesssor cases, multiple jobs or processes may be active at one time and it is the responsibility of the operating system to schedule their expiation and to allocate resources. A user may construct application that use multiple jobs or process may be active at one time, and it is the responsibility of the operating system to schedule their execution and to allocate resources. A user may construct application that use multiple processes or multiple threads within processes without regard to whether a simple processor or multiple processor will be available. Thus a multi processor operating system must provide all the functionality of a multi programming system plus additional features to accord ate multiple processor. Among the key design issues:

- **Simultaneous concurrent processes:** Operating system routines need to be reentrant to allow several processor to execute the case is code simultaneously. With multiple processors executing the same or different parts of the operating system, operating system tables and management structures must be managed properly to avoid dead lock or invalid operations.
- **Scheduling:** Any processor may perform scheduling, so conflict must be avoided. The scheduler must assign ready processes to available processors.

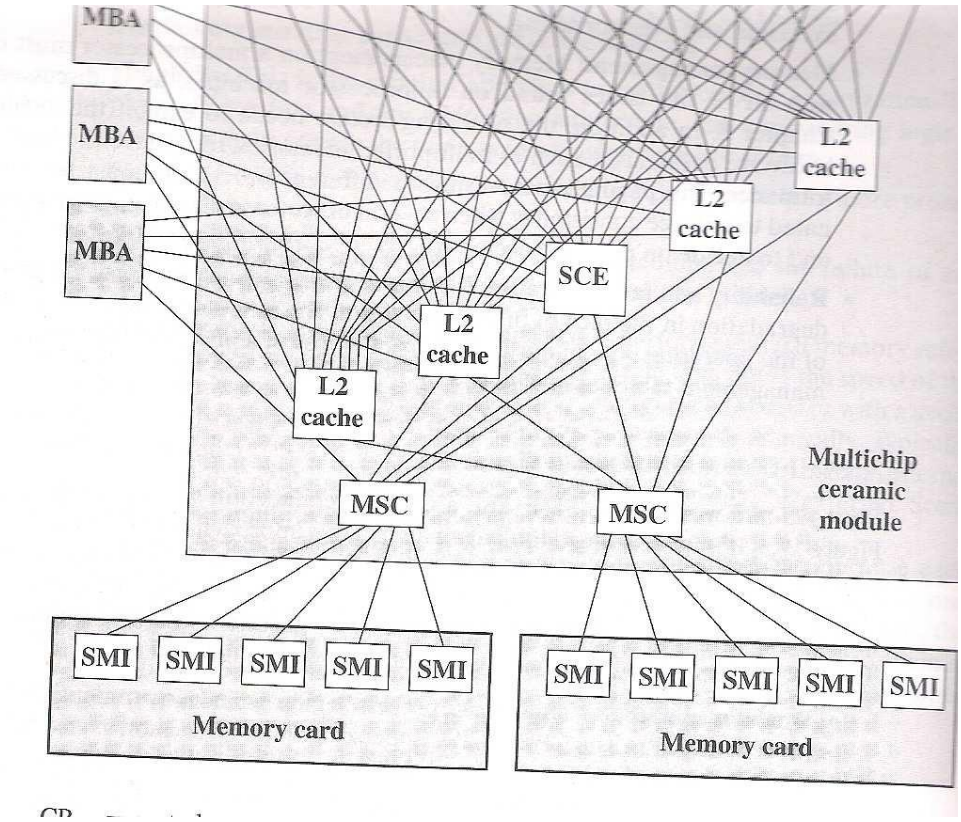
- **Synchronization:** With multiple active processes having potential accesses to shared address spaces or shared I/O resources, care must be taken to provide effective. Synchronization is a facility that enforces mutual exclusion and event ordering.
- **Memory management:** Memory management on a multi processor must deal with all of the issues found on unit processor machines. In addition the operating system needs to exploit the available hardware parallelism, such as multi ported memories, to achieve the best performance. The paging mechanism on different processors shares a page replacement.
- **Reliability and fault tolerance:** The operating system should provide graceful degradation influence of processor failure. The scheduler and other portions of the operating system must recognize the loss of a processor and restructure management tables accordingly.

3.2 A MAIN FRAME SMP

Most pc and work station smps use a bus interconnection strategy as depicted in figure 3.1.1. It is instructive to look at an alternative approach, which is used for qa recent implementation of the IBM series main frame family of systems spans a range from a uniprocessor with one main memory card to a high- end system with 48 processors and 8 memory cards. The key components of the configuration are shown in figure 3.3.2

- **Dual:** core processor chip: Each processor chip includes two identical central processor, in which most of the instructions are hard wired and the rest are executed by vertical micro code. Each CP includes 9256- KBL 1 instruction cache and a 256- KB data cache.
- **L2 cache:** Each L2 caches are arranged in clusters of five, with each cluster supporting eight processor chips and providing access to the entire main memory space.
- **System control element (SCE**): The SCE arbitrates system communication and has a central role in maintaining cache coherence.
- **Main store control (MSC):** The MSCs interconnect the L2 caches and the main memory.
- **Memory card:** Each card holds 32 GB of memory. The maximum configurable memory consists of 8 memory cards for a total of 256 GB. Memory cards interconnect to the MSC via synchronous memory interfaces (SMIs)

- **Memory has adapter (MBA):** The MBA provides an interface to various types of I/O channels. Traffic to/from the channels goes directly to the L2 cache.
- The microprocessor in the 2990 is relatively uncommon compared with other modern processors because although. It is superscalar it executes instructions in strict architectural order



4.0 CONCLUSION

The term SMP refers to a computer hardware architecture and also to the operating system behaviour that reflects that architecture. It can be defined as a stands alone computer system with the following characteristics.

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme such that memory aces time is approximately the same for each process.

3. All processors share access to I/O devices either through the same channels or through different channels that provide paths to the same device.

4. All process can perform the same function (hence the term symmetric).

5. The system is controlled by an integrated operating system that provide interaction between processors and their programs at the job task file and data element levels.

Points 1 to 4 should be self-explanatory. Point 5 illustrates one of the contrasts with a loosely coupled multiprocessing system, such as a cluster. In the latter, the physical unit of interaction is usually a message or complete file. In an SNIP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

The operating system of an SMP schedules processes or threads across all of the processors. An SMP organization has a number of potential advantages over a uniprocessor organization.

5.0 SUMMARY

- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.

- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.

- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronization among processors.

6.0 TUTOR MARKED ASSIGNMENT

1. What are some of the potential advantages of an SMP compared with a uniprocessor?
2. What are the chief characteristics of an SMP?
3. What are some of key operating system design issues for an SMP?

7.0 References/ Further reading

Milenkovic, A. “Achieving High Performance in Bus- Based shared memory multiprocessors” IEEE concurrency, July- September 2000

UNIT 3: MULTI THREADING AND CHIP MULTI PROCESSORS

CONTENT

1.0 INTRODUCTION

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as MIPS rate = *f* x IPC where *f is* the processor clock frequency, in MHz, and IPC (instructions per cycle) is the average number of instructions executed per cycle. Accordingly, designers have pursued the goal of increased performance on two fronts: increasing clock frequency and increasing the number of instructions executed or, more properly, the number of instructions that complete during a processor cycle.

An alternative approach, which allows for a high degree of instruction-level parallelism without increasing circuit complexity or power consumption, is called multithreading. In essence, the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel.

The variety of specific multithreading designs, realized in both commercial systems and experimental systems, is vast.

2.0 OBJECTIVES

At the end of this unit you should be able to:

- Explain multi threading and chip multi processor.
- Discuss implicit and explicit multi threading.
- Understand the four principal approaches to multithreading.

3.1 IMPLICIT AND EXPLICIT MULTITHREADING

The concept of thread used in discussing multi-thread processors may not be the same as the concept of software threads in a multi programmed operating system. It will be useful to define terms briefly:

- Process: An instance of a progam running on a computer. A process embodies two key characteristics.

- Resource ownership: A process includes a virtual address space to hold the process image; the process image is the collections of program, data, stand attributes that define the process. From time to time, a process may allocate control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.

- Scheduling/execution: The execution of a process follows an execution (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.

Process switch: An operation that switches the processor from one process to another, by saving all the process control data, registers, and other information for the first and replacing them with the process information for the second.

Thread: A dispatch able unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.

Thread switch: The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.

Thus, a thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership. The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch. Traditional operating systems, such as earlier versions of UNIX, did not support threads. Most modern operating systems, such as Linux, other versions of UNIX, and Windows, do support thread. A distinction is made between user-level threads, which are visible to the application program, and kernel-level threads, which are visible only to the operating system. Both of these may be referred to as explicit threads, defined in software.

All of the commercial processors and most of the experimental processors so far have used explicit multithreading. These systems concurrently execute instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. Implicit multithreading refers to the concurrent execution of multiple threads extracted from a single sequential program. These implicit threads may be defined either statically by the compiler or dynamically by the hardware.

3.2 APPROACHES TO EXPLICIT MULTI THREADING

At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently. The designs differ in the amount and type of additional hardware used to support concurrent thread execution. In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and may use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques. What is achieved is thread-level parallelism, which may provide for greatly improved performance when married to instruction-level parallelism. Broadly speaking, there are four principal approaches to multithreading:

Interleaved multithreading: This is also known as fine-grained multithreading. The processor deals with two or more thread contexts at a time, switching from one thread to another at each clock cycle. If a thread is blocked because of data dependencies or memory latencies. that thread is skipped and a ready thread is executed.

Blocked multithreading: This is also known as coarse-grained multithreading. The instructions of a thread are executed successively until an event occurs that may cause delay, such as a cache miss. This event induces a switch to another thread. This approach is effective on an in-order processor that would stall the pipeline for a delay event such as a cache miss.

- ✓ **Simultaneous multithreading (SMT):** Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. This combines the wide superscalar instruction issue capability with the use of multiple thread contexts.
- ✓ **Chip multiprocessing:** In this case, the entire processor is replicated on a single chip and each processor handles separate threads. The advantage of this approach is that the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design. This is referred to as multicore;

For the first two approaches, instructions from different threads are not executed simultaneously. Instead, the processor is able to rapidly switch from one thread to another, using a different set of registers and other context information. This results in a better utilization of the processor's execution resources and avoids a large penalty due to cache misses and other latency events. The SMT approach involves true simultaneous execution of instructions from different threads, using replicated execution resources. Chip multiprocessing also enables simultaneous execution of instructions from different threads.

- **Single-threaded scalar:** This is the simple pipeline found in traditional Riand CISC machines, with no multithreading.
- **Interleaved multithreaded scalar**: This is the easiest multithreading approach to implement. By switching from one thread to another at each clock cycle. The pipeline stages can be kept fully occupied, or close to fully occupied. The hardware must be capable of switching from one thread context to another between cycles.

3.3 EXAMPLE SYSTEMS

Symmetric multi threading. This is the processor has a super scalar architecture and can issue instructions from one or both threads in parallel. At the end of the pipeline, separate thread resources are needed to commit the instructions.

4.0 CONCLUSION

A multi cone computer also known as a chip multi processor, combines two ormore processor (called cores) on a single pice of silicon (called adie). Another organizational design decision in a multi core will be superscalar or will implement simultaneous multi threading (SMI)

5.0 SUMMARY

An related design scheme is to replicate some of the components of a single processor so that the processor con execute multiple threads concurrently; thus is known as a multi thread processor. When more than one processor are implemented on a single chip the configuration is referred to as chip multi processing.

6.0 TUTOR MARKED ASSIGNMENT

1. Explain multi threading
2. List and briefly explain four principal approaches to multi threading

7.0 References/ further reading

Ungerere, T. Rubic B and sile J" Multi threaded processors the computer journal No 3 2002

UNIT 4: VECTOR COMPUTATION

1.0 INTRODUCTION

Although the performance of mainframe general-purpose computers continues to improve relentlessly, there continue to be applications that are beyond the reach of the contemporary mainframe. There is a need for computers to solve mathematical problems of physical processes, such as occur in disciplines including aerodynamics, seismology, meteorology, and atomic, nuclear, and plasma physics.

2.0 OBJECTIVES

At the end of this unit, you should be able to

- Understand that super computers are optimized for vector computation.
- Explain the contrast between main frame and super computers as it relates to vectors computation.

4.0 CONCLUSION

Supercomputers were developed to handle these types of problems. These machines are typically capable of billions of floating-point operations per second. In contrast to mainframes, which are designed for multiprogramming and intensive I/O. the supercomputer is optimized for the type of numerical calculation just described. The supercomputer has limited use and, because of its price tag, a limited market.

5.0 SUMMARY

Comparatively few of these machines are operational, mostly at research centers and some government agencies with scientific or engineering functions. As with other areas of computer technology, there is a constant demand to increase the performance of the supercomputer. Thus, the technology and performance of the supercomputer continues to evolve.

There is another type of system that has been designed to address the need to vector computation, referred to as the array processor. Although a supercomputer optimized for vector computation, it is a general-purpose computer, capable of handling scalar processing and general data processing tasks. Array processors do not include scalar processing; they are configured as peripheral devices by both mainframe and minicomputer users to run the vectorized portions of programs.

3.2 IBM 3090 VECTOR FACILITY

A good example of a pipelined ALU organization for vector processing is the vector facility developed for the IBM 370 architecture and implemented on the high-end 3090 series [PADE88, TUCK87]. This facility is an optional add-on to the basic system but is highly integrated with it. It resembles vector facilities found on supercomputers, such as the Cray family.

The IBM facility makes use of a number of vector registers. Each register is actually a bank of scalar registers. To compute the vector sum C = A + B, the vectors A and B are loaded into two vector registers. The data from these registers are passed through the ALU as fast as possible, and the results are stored in a third vector register. The computation overlap, and the loading of the input data into the registers in a block, results in a significant speeding up over an ordinary ALU operation.

The IBM vector architecture, and similar pipelined vector ALUs. provides increased performance over loops of scalar arithmetic instructions in three ways:

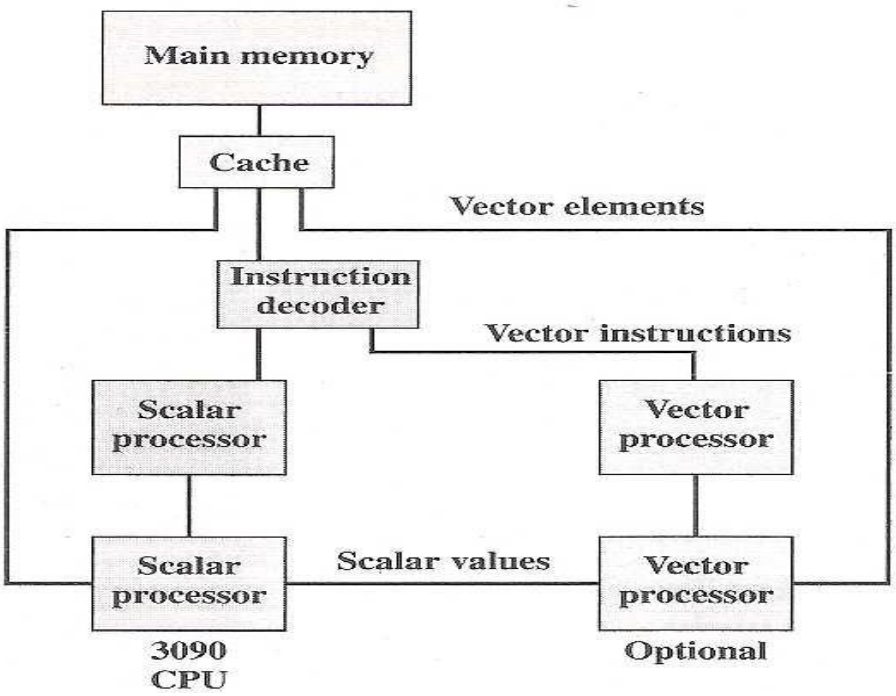
The fixed and predetermined structure of vector data permits housekeeping instructions inside the loop to be replaced by faster internal (hardware or microcode) machine operations.

Data-access and arithmetic operations on several successive vector elements can proceed concurrently by overlapping such operations in a pipelined design_ or by performing multiple-element operations in parallel.

The use of vector registers for intermediate results avoids additional storage reference.

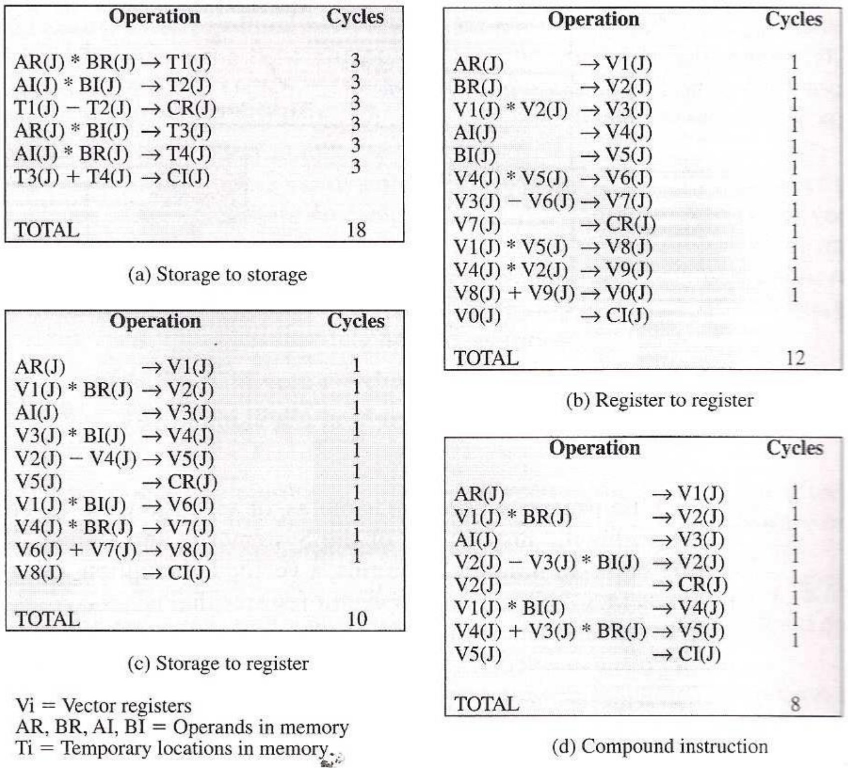
Figure 3.2.1 shows the general organization of the vector facility. Although the vector facility is seen to be a physically separate add-on to the processor, its architecture is an extension of the System/370 architecture and is compatible with it. The vector facility is integrated into the System J370 architecture in the following ways:

- Existing System/370 instructions are used for all scalar operations.
- Arithmetic operations on individual vector elements produce exactly the sale result as do corresponding System/370 scalar instructions. For example, or design decision concerned the definition of the result in a floating-point DIVIDE operation. Should the result be exact, as it is for scalar floating-point, division, or should an approximation be allowed that would permit higher speed implementation but could sometimes introduce an error in one or mare low-order bit positions? The decision was made to uphold complete compatibility with the System/370 architecture at the expense of a minor performance degradation.
- Vector instructions are interruptible, and their execution can be resumed for the point of interruption after appropriate action has been taken, in a manne-compatible with the System/370 program-interruption scheme.



- Arithmetic exceptions are the same as, or extensions of, exceptions for the scalar arithmetic instructions of the System/370, and similar fix-up routines can be used. To accommodate this, a vector interruption index is employed that indicates the location in a vector register that is affected by an exception (e.g., overflow). Thus, when execution of the vector instruction resumes, the proper place in a vector register is accessed.
- Vector data reside in virtual storage, with page faults being handled in a standard manner.
- This level of integration provides a number of benefits. Existing operating systems can support the vector facility with minor extensions. Existing application programs, language compilers, and other software can be run unchanged. Software that could take advantage of the vector facility can be modified as desired.

A key issue in the design of a vector facility is whether operands are located in registers or memory. The IBM organization is referred to as *register to register,* because the vector operands, both input and output, can be staged in vector registers.



Operation	Cycles
AR(J) * BR(J) → T1(J)	3
AI(J) * BI(J) → T2(J)	3
T1(J) − T2(J) → CR(J)	3
AR(J) * BI(J) → T3(J)	3
AI(J) * BR(J) → T4(J)	3
T3(J) + T4(J) → CI(J)	3
TOTAL	18

(a) Storage to storage

Operation	Cycles
AR(J) → V1(J)	1
BR(J) → V2(J)	1
V1(J) * V2(J) → V3(J)	1
AI(J) → V4(J)	1
BI(J) → V5(J)	1
V4(J) * V5(J) → V6(J)	1
V3(J) − V6(J) → V7(J)	1
V7(J) → CR(J)	1
V1(J) * V5(J) → V8(J)	1
V4(J) * V2(J) → V9(J)	1
V8(J) + V9(J) → V0(J)	1
V0(J) → CI(J)	1
TOTAL	12

(b) Register to register

Operation	Cycles
AR(J) → V1(J)	1
V1(J) * BR(J) → V2(J)	1
AI(J) → V3(J)	1
V3(J) * BI(J) → V4(J)	1
V2(J) − V4(J) → V5(J)	1
V5(J) → CR(J)	1
V1(J) * BI(J) → V6(J)	1
V4(J) * BR(J) → V7(J)	1
V6(J) + V7(J) → V8(J)	1
V8(J) → CI(J)	1
TOTAL	10

(c) Storage to register

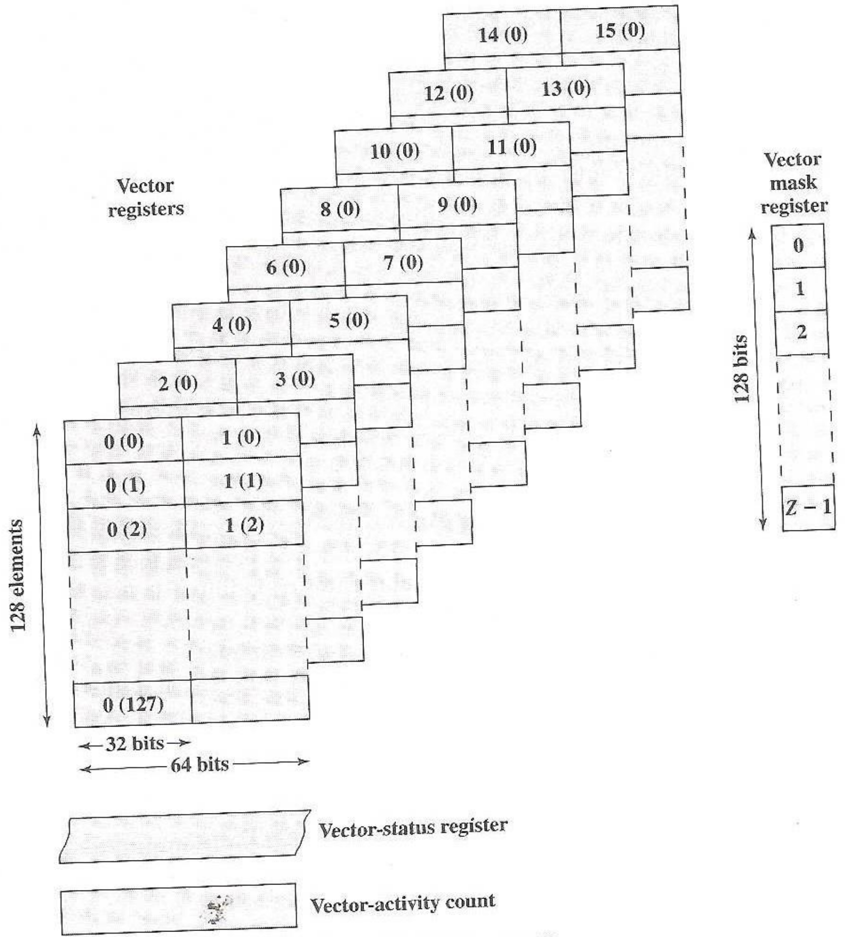
Operation	Cycles
AR(J) → V1(J)	1
V1(J) * BR(J) → V2(J)	1
AI(J) → V3(J)	1
V2(J) − V3(J) * BI(J) → V2(J)	1
V2(J) → CR(J)	1
V1(J) * BI(J) → V4(J)	1
V4(J) + V3(J) * BR(J) → V5(J)	1
V5(J) → CI(J)	1
TOTAL	8

(d) Compound instruction

Vi = Vector registers
AR, BR, AI, BI = Operands in memory
Ti = Temporary locations in memory

This approach is also used on the Cray supercomputer. An alternative approach, used on Control Data machines, is to obtain operands directly from memory. The main disadvantage of the use of vector registers is that the programmer or compiler must take them into account for good performance. For example, suppose that the length of the vector registers is K and the length of the a minor performance, vectors to be processed is $N > K$. In this case, a vector loop must be performed, in which the operation is performed on K elements at a time and the loop is repeated N/K times. The main advantage of the vector register approach is that the can be resumed from operation is decoupled from slower main memory and instead takes place primarily taken, in a marine with registers.

The speedup that can be achieved using registers is demonstrated in F17.20. The FORTRAN routine multiplies vector A by vector B to produce C, where each vector has a real part (AR, BR, CR) and an imaginary part (Ai. CI). The 3090 can perform one main-storage access per processor, or clock.(either read or write); has registers that can sustain two accesses for reading one for writing per cycle; and produces one result per cycle in its arithmetic. Let us assume the use of instructions that can specify two source operands result. Part a of the figure shows that, with memory-to-memory instructions iteration of the computation 'requires a total of 18 cycles.



register architecture (part b), this time is reduced to 12 cycles. Of course, with register-to-register operation, the vector quantities must be loaded into the vector registers prior to computation and stored in memory afterward. For large vectors, this fixed penalty is relatively small. Figure 17.20c shows that the ability to specify both storage and register operands in one instruction further reduces the time to 10 cycles per iteration. This latter type of instruction is included in the vector architecture.[5]

Figure 17.21 illustrates the registers that are part of the IBM 3090 vector facility.There are sixteen 32-bit vector registers. The vector registers can also be coupled to form eight 64-bit vector registers. Any register element can hold an integer or floating-point value. Thus, the vector registers may be used for 32-bit and 64-bit integer values, and 32-bit and 64-bit floating-point values.

6.0 TUTOR MARKED ASSIGNMENT

1. What are the chief characteristics of SMP

2. Produce a vectorized version of the following programs

```
DO20I=1,N
B(I,1)-0
AL(I)=A(I)+B(I,J)X(I,J)
20 CONTINUE
20 CONTINUE
```

7.0 REFERENCES/ FURTHER READING

Tomascivic, M amd Multinsvic, V. The cache coherence problem in shared memory Multiprocessors. Hardware solutions Los Alamitos, C. A: IEEE computers society press, 1993.

MODULE 4 REDUCED INSTRUCTION SET COMPUTERS

UNIT 1: INSTRUCTIONS EXECUTION CHARACTERS
UNIT2: REDUCED INSTRUCTION SET ARCHITECTURE

CONTENT

1.0 INTRODUCTION

Studies of the execution behaving high- level language programs have provided guidance's in designing a new type of processor architecture. The reduced instruction set computer (RISC)

2.0 OBJECTIVES

At the end of this unit you should be able to

- Understand the operation of RISC machines
- Understand the number of parameters and variables a procedure deals with and the depth of nesting.

3.1 OPERATIONS

There is quite good agreement in the results of this mixture of languages and applications. Assignment statements predominate. suggesting that the simple movement - of data is of high importance. There is also a preponderance of conditional statements (IF. LOOP). These statements are implemented in machine language wit= some sort of compare and branch instruction. This suggests that the sequence control mechanism of the instruction set is important.

These results are instructive to the machine instruction set designer, indicator, which types of statements occur most often and therefore should be supported in optimal" fashion. However these results do not reveal which statements use the most time in the execution of a typical program. That is, given a compiled machine language program, which statements in the source language cause the execution o= the most machine-language instructions?

To get at this underlying phenomenon, the Patterson programs [PATT82aj. described in Appendix 4A. were compiled on the VAX, PDP-11, and MotorolL 68000 to determine the average number of machine instructions and memory references per statement type. The second and third columns in Table 13.2 show the relative frequency of occurrence of various HLL instructions in a variety of programs; the data were obtained by observing the occurrences in running programs rather than just the number of times that statements occur in the source code.

3.2 OPERANDS

Much less work has been done on the occurrence of types of operands, despite the importance of this topic. There are several aspects that are significant.

The Patterson study already referenced [PATT82a] also looked at the dynamic frequency of occurrence of classes of variables (Table 13.3). The results, consistent between Pascal and C programs, show that the majority of references are to simple scalar variables. Further, more than 80 % of the scalars were local (to the procedure) variables. In addition, references to arrays/structures require a previous reference to their index or pointer, which again is usually a local scalar. Thus, there is a preponderance of references to scalars, and these are highly localized.

The Patterson study examined the dynamic behavior of HLL programs, independent of the underlying architecture. As discussed before, it is necessary to deal with actual architectures to examine program behavior more deeply. One study, [LUND77], examined DEC-10 instructions dynamically and found that each instruction on the average references 0.5 operand in memory and 1.4 registers. Similar results are reported in [HUCK83] for C, Pascal, and FORTRAN programs on S/370, PDP-11, and VAX. Of course, these figures depend highly on both the architecture and the compiler, but they do illustrate the frequency of operand accessing.

These latter studies suggest the importance of an architecture that lends itself to fast operand accessing, because this operation is performed so frequently. The Patterson study suggests that a prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (Table 13.2) suggests that these are the most time-consuming operations in compiled HLL programs. Thus, it will be profitable to consider ways of implementing these operations efficiently. Two aspects are significant: the number of parameters and variables that a procedure deals with, and the depth of nesting.

Tanenbaum's study [TANE78] found that 98% of dynamically called procedures were passed fewer than six arguments and that 92% of them used fewer than six local scalar variables. Similar results were reported by the Berkeley RISC team [KATE83], as shown in Table 13.4. These results show that the number of words required per procedure activation is not large., The studies reported earlier indicated that a high proportion of operand references is to local scalar variables. These studies show that those references are in fact confined to relatively few variables.

3.3 IMPLICATIONS

A number of groups have looked at results such as those just reported and have concluded that the attempt to make the instruction set architecture close to HLLs is not the most effective design strategy. Rather, the HLLs can best be supported by optimizing performance of the most time-consuming features of typical HLL programs.

Generalizing from the work of a number of researchers, three elements emerge that, by and large, characterize RISC architectures. First, use a large number of registers or use a compiler to optimize register usage. This is intended to optimize operand referencing. The studies just discussed show that there are several references per HLL instruction and that there is a high proportion of move (assignment) statements. This, coupled with the locality and predominance of scalar references, suggests that performance can be improved by reducing memory references at the expense of more register references. Because of the locality of these references, an expanded register set seems practical.

Second, careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call instructions, a straightforward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are prefetched but never executed.

Finally, a simplified (reduced) instruction set is indicated. This point is not as obvious as the others, but should become clearer in the ensuing discussion.

4.0 CONCLUSION

Assignment statements predominate, suggesting that the simple movement of data should be optimized. There are also many IF and LOOP instructions, which suggest that the underlying sequence control mechanism needs to be optimized to permit efficient pipelining. Studies of operand reference patterns suggest that it should be possible to enhance, performance by keeping A moderate number of operands in registers.

5.0 SUMMARY

The simple instruction set of a RISC lends itself to efficient pipelining because there are fewer and more predictable operations performed per instruction. Other instruction to improve pipeline efficiency.

6.0 Tutor marked assignment

1. What is a delayed branch?

7.0 REFERENCES/ FURTHER READING

Patterson, D “Reduced instruction set computers communications of the ACM, January 1985.

UNIT 2: REDUCED INSTRUCTION SET ARCHITECTURE

CONTENT

1.0 INTRODUCTION

This is the focus of this unit. The RISC architecture is a dramatic departure from the historical trend in processor architecture. An analysis of the RISC architecture brings into focus many of the important issues in computer organization and architecture.

2.0 OBJECTIVES

At the end of this unit, you should be able to understand the pitfalls in the CISC approach in companion to RISC.

3.1 Why CISC

In this section, we look at some of the general characteristics of and the motivation for a reduced instruction set architecture. Specific examples will be seen later in this chapter. We begin with a discussion of motivations for contemporary complex instruction set architectures.

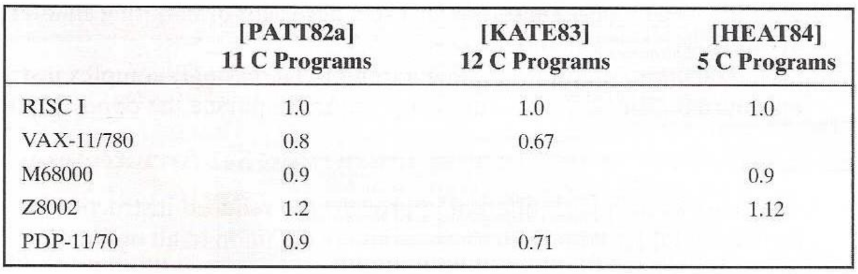
We have noted the trend to richer instruction sets, which include a larger number of instructions and more complex instructions. Two principal reasons have motivated this trend: a desire to simplify compilers and a desire to improve performance. Underlying both of these reasons was the shift to HLLs on the part of programmers; architects attempted to design machines that provided better support for HLLs.

It is not the intent of this chapter to say that the CISC designers took the wrong direction. Indeed, because technology continues to evolve and because architectures exist along a spectrum rather than in two neat categories, a black-and-white assessment is unlikely ever to emerge. Thus, the comments that follow are simply meant to point out some of the potential pitfalls in the CISC approach and to provide some understanding of the motivation of the RISC adherents.

The first of the reasons cited, compiler simplification, seems obvious. The task of the compiler writer is to generate a sequence of machine instructions for each FILL statement. If there are machine instructions that resemble HLL statements, this task is simplified. This reasoning has been disputed by the RISC researchers ([HENN82]. [RAIJI83], [PATT82b]). They have found that complex machine instructions are often hard to exploit because the compiler must find those cases that exactly fit the construct. The task of optimizing the generated code to minimize code size, reduce instruction execution count, and enhance pipelining is much more difficult with a complex instruction set. As evidence of this, studies cited earlier in this chapter indicate that most of the instructions in a compiled program are the relatively simple ones.

The other major reason cited is the expectation that a CISC will yield smaller. faster programs. Let us examine both aspects of this assertion: that programs will be smaller and that they will execute faster.

There are two advantages to smaller programs. First, because the program takes up less memory, there is a savings in that resource. With memory today being so inexpensive, this potential advantage is no longer compelling. More important smaller programs should improve performance, and this will happen in two ways. First, fewer instructions means fewer instruction bytes to be fetched. Second, in a paging environment, smaller programs occupy fewer pages, reducing page faults.



	[PATT82a] 11 C Programs	[KATE83] 12 C Programs	[HEAT84] 5 C Programs
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

The problem with this line of reasoning is that it is far from certain that a CISC program will be smaller than a corresponding RISC program. In many cases, the CISC program, expressed in symbolic machine language, may be *shorter (i.e.,* fewer instructions), but the number of bits of memory occupied may not be noticeably *smaller.* Table 13.6 shows results from three studies that compared the size of compiled C programs on a variety of machines, including RISC I, which has a reduced instruction set architecture. Note that there is little or no savings using a CISC over a RISC. It is also interesting to note that the VAX, which has a much more complex instruction set than the PDP-11, achieves very little savings over the latter. These results were confirmed by IBM researchers [RADI83], who found that the IBM 801 (a RISC) produced code that was 0.9 times the size of code on an IBM S/370. The study used a set of PL/I programs.

There are several reasons for these rather surprising results. We have already noted that compilers on CISCs tend to favor simpler instructions, so that the conciseness of the complex instructions seldom comes into play. Also, because there are more instructions on a CISC, longer opcodes are required, producing longer instructions. Finally, RISCs ten`s to emphasize register rather than memory _references. and the former require fewer bits. An example of this last effect is discussed presently.

So the expectation that a CISC will produce smaller programs, with the attendant advantages, may not be realized. The second motivating _factor for increasingly complex instruction sets was that instruction execution would be faster. It seems to make sense that a complex HLL operation will execute more quickly as a single machine instruction rather than as a series of more primitive instructions. However, because of the bias toward the use of those simpler instructions, this may not be so. The entire control unit must be made more complex, and/or the microprogram control store must be made larger, to accommodate a richer instruction set. Either factor increases the execution time of the simple instructions.

In fact, some researchers have found that the speedup iii the execution of complex functions is due not so much to the power of the complex machine instructions as to their residence in high-speed control store [RADI83]. In effect, the control store acts as an instruction cache. Thus, the hardware architect is in the position of trying to determine which subroutines or functions will be used most frequently and assigning those to the control store by implementing them in microcode. The results have been less than encouraging. On S/390 systems, instructions such as Translate and Extended-Precision-Floating-Point-Divide reside in high-speed storage, while the sequence involved in setting up procedure calls or initiating an interrupt handler are in slower main memory.

Thus, it is far from clear that a trend to increasingly complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

3.2 CHARACTERISTICS OF REDUCED INSTRUCTION SET ARCHITECTURES

Although a variety of different approaches to reduced instruction set architecture have beers taken, certain characteristics are common to all of them:

- One instruction per cycle
- Register-to-register operations
- Simple addressing modes
- Simple instruction formats

Here, we provide a brief discussion of these characteristics. Specific examples are explored later in this chapter.

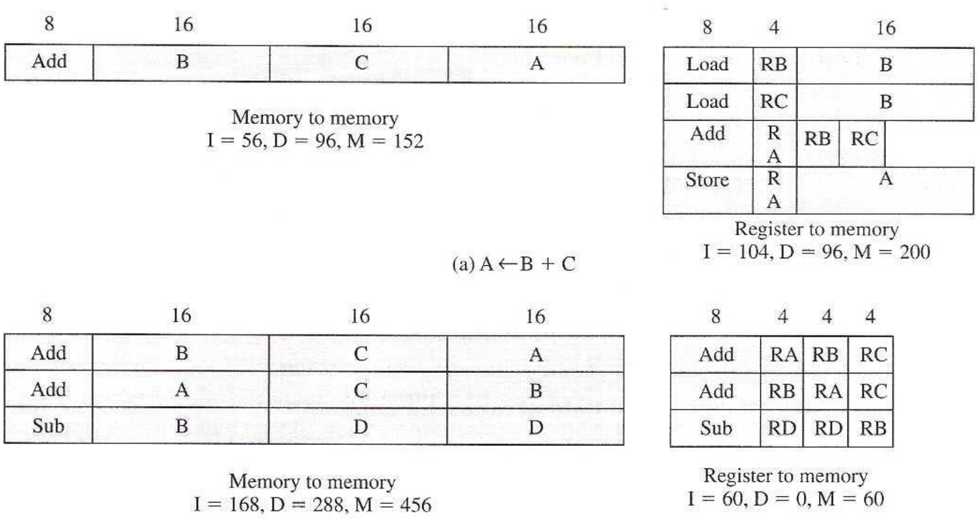
The first characteristic listed is that there is one machine instruction per machine cycle. *A machine cycle* is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register. Thus, RISC machine instructions should be no more complicated than, and execute about as fast as, microinstructions on CISC machines (discussed in Part Four). With simple, one-cycle instructions, there is little or no need for microcode; the machine instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on other machines, because it is not necessary to access a microprogram control store during instruction execution.

A second characteristic is that most operations should be register to register. with only simple LOAD and STORE operations accessing memory. This design feature simplifies the instructio4Lset and therefore the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g., integer add, add with carry); the VAX has 25 different ADD instructions. Another benefit " that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage.

This emphasis on register-to-register operations is notable for RISC design Contemporary CISC machines provide such instructions but also include memory to memory and mixed register/memory operations. Attempts to compare these approaches were made in the 1970s, before the appearance of RISCs. illustrates the approach taken. Hypothetical architectures were evaluated on program size and the number of bits of memory traffic. Results such as this one led researcher to suggest that future architectures should contain no registers a: [MYER78]. One wonders what he would have thought, at the time, of the RISC machine once produced by Pyramid, which contained no less than 528 registers!

What was missing from those studies was a recognition of the frequent access to a small number of local scalars and that, with a large bank of registers or an optimizing compiler, most operands could be kept in registers for a long periods of time.
Thus, Figure 13.5b may be a fairer comparison.

A third characteristic is the use of simple addressing modes. Almost all RISC instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included. Other, more complex modes can be synthesized in software from the simple ones. Again, this design feature simplifies the instruction set and the control unit.



8	16	16	16
Add	B	C	A

Memory to memory
I = 56, D = 96, M = 152

8	4	16
Load	RB	B
Load	RC	B
Add	RA	RB RC
Store	RA	A

Register to memory
I = 104, D = 96, M = 200

(a) A←B + C

8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

Memory to memory
I = 168, D = 288, M = 456

8	4	4	4
Add	RA	RB	RC
Add	RB	RA	RC
Sub	RD	RD	RB

Register to memory
I = 60, D = 0, M = 60

(b) A← B + C; B←A + C; D←D − B

A final common characteristic is the use of **simple instruction formats.** Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit. Instruction fetching is optimized because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

Taken together, these characteristics can be assessed to determine the potential performance benefits of the RISC approach. A certain amount of "circumstantial evidence" can be presented. First, more effective optimizing compilers can be developed. With more-primitive instructions, there are more opportunities for moving functions out of loops, reorganizing code for efficiency, maximizing register utilization, and so forth. It is even possible to compute parts of complex instructions at compile time. For example, the S/390 Move Characters (MVC) instruction moves a string of characters from one location to another. Each time it is executed, the move will depend on the length of the string, whether and in which direction the locations overlap, and what the alignment characteristics are. In most cases, these will all be known at compile time. Thus, the compiler could produce an optimized sequence of primitive instructions for this function.

A second point, already noted, is that most instructions generated by a compiler are relatively simple anyway. It would seem reasonable that a control unit built specifically for those instructions and using little or no microcode could execute them faster than a comparable CISC.

A third point relates to the use of instruction pipelining. RISC researchers feel that the instruction pipelining technique can be applied much more effectively with a reduced instruction set. We examine this point in some detail presently.

A final, and somewhat less significant, point is that RISC processors are more responsive to interrupts because interrupts are checked between rather elementary operations. Architectures with complex instructions either restrict interrupts to instruction boundaries or must define specific interruptible points and implement mechanisms for restarting an instruction.

The case for improved performance for a reduced instruction set architecture is strong, but one could perhaps still make an argument for CISC. A number of studies have been done but not on machines of comparable technology and power. Further, most studies have not attempted to separate the effects of a reduced instruction set and the effects of a large register file. The "circumstantial evidence," however, is suggestive.

3.3 CISC VERSUS RICS CHARACTERISTICS

After the initial enthusiasm for RISC machines, there has been a growing realization that (1) RISC designs may benefit from the inclusion of some CISC features and that (2) CISC designs may benefit from the inclusion of some RISC features. The result is that the more recent RISC designs, notably the PowerPC, are no longer "pure" RISC and the more recent CISC designs, notably the Pentium II and later Pentium models, do incorporate some RISC characteristics.

An interesting comparison in [MASH95] provides some insight into this issue. Table 13.7 lists a number of processors and compares them across a number of characteristics. For purposes of this comparison, the following are considered typical of a classic RISC:

1. A single instruction size.
2. That size is typically 4 bytes.
3. A small number of data addressing modes, typically less than five. This paramet:is difficult to pin down. In the table, register and literal modes are not count and different formats with different offset sizes are counted separately.
4. No indirect addressing that requires you to make one memory access to get address of another operand in memory.
5. No operations that combine load/store with arithmetic (e.g., add from mem,-add to memory).
6. No more than one memory-addressed operand per instruction.
7. Does not support arbitrary alignment of data for load/store operations.
8. Maximum number of uses of the memory management unit (MMU) for a c: address in an instruction.
9. Number of bits for integer register specified equal to five or more. This means that at least 32 integer registers can be explicitly referenced at a time.
10. Number of bits for floating-point register specifier equal to four or more. This means that at least 16 floating-point registers can be explicitly referenced at a time.

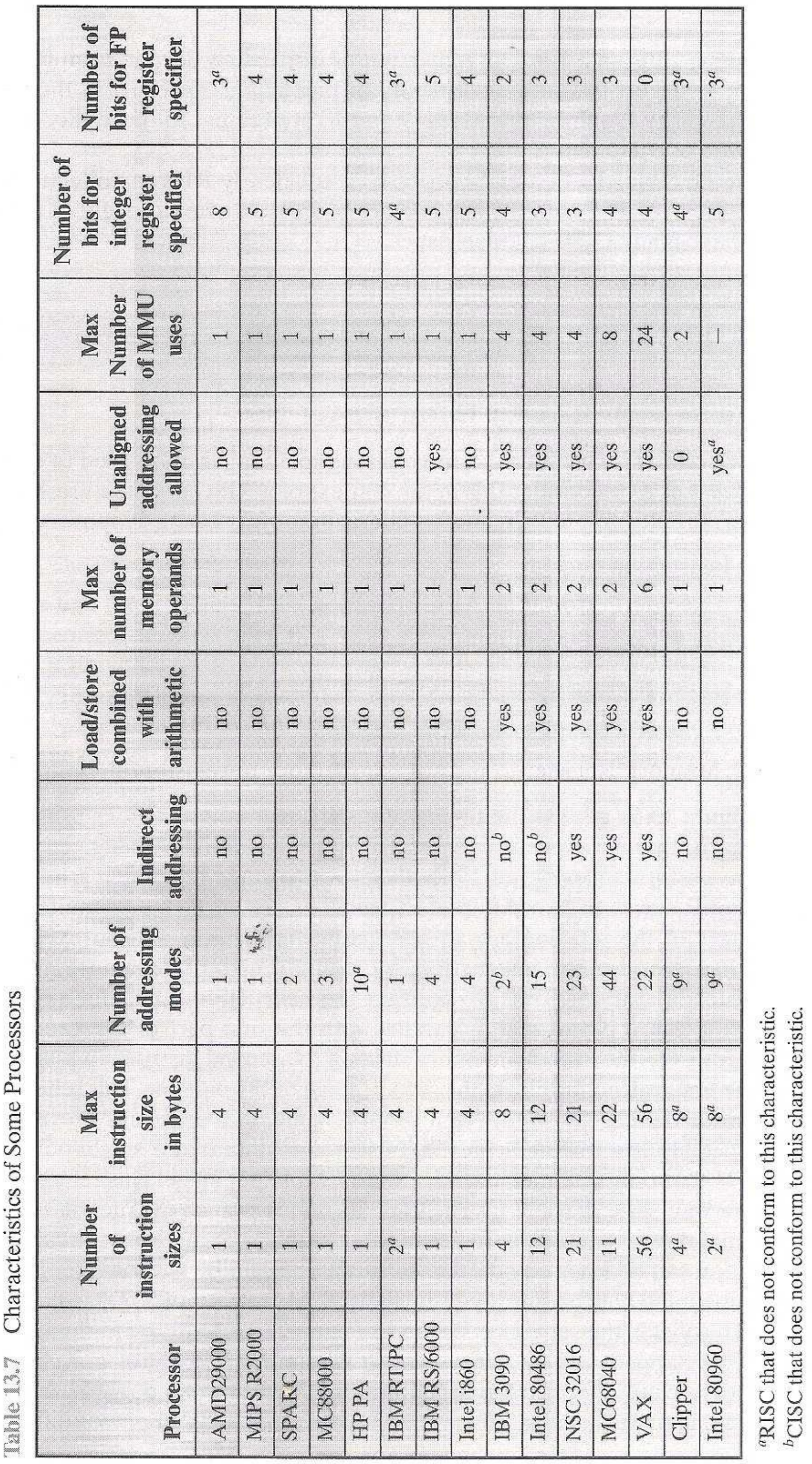


Table 13.7 Characteristics of Some Processors

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

[a] RISC that does not conform to this characteristic.

[b] CISC that does not conform to this characteristic.

Items 1 through 3 are an indication of instruction decode . -:: - through 8 suggest the ease or difficulty of pipelining, especial L_, virtual memory requirements. Items 9 and 10 are related to the advantage of compilers.

In the table, the first eight processors are clearly RISC are five are clearly CISC, and the last two are processors often thou in fact have many CISC characteristics.

UNIT 3 : RISC PIPELINING

1.0 introductio

Instruction pipelining is often used to enhance performances and can be improved further by permitting two memoryaccesses per stage.

2.0 objectives

At the end of this unit, you should be able to

-Explain the stages involve in an instruction cycle

- Explain the stages required for LOAD and STORE operations

3.1 PIPELINIG WITH REGULAR INSTRUCTIONS

Instruction pipelining is often to enhance performance. Let us reconsider this in the context of a RISC architecture. Most instructions are register to register, and an instruction cycle has the follow.

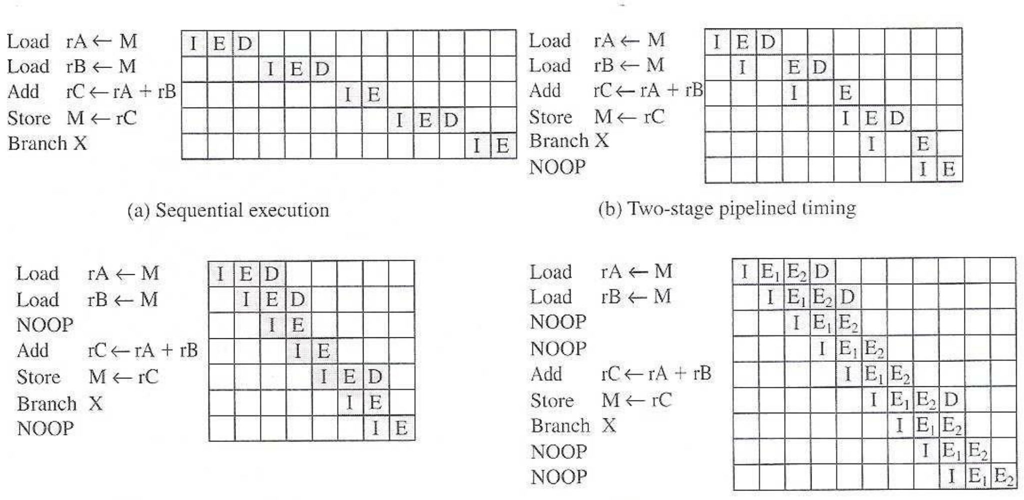
- I: Instruction fetch.
- E: Execute. Performs an ALU operation with register input any

For load and store operations, three stages are required:

I: Instruction fetch.
E: Execute. Calculates memory address
D: Memory. Register-to-memory or memory-to-register operates.

Figure 3.1.A depicts the timing of a sequence of instructions using. Clearly, this is a wasteful process. Even very simple pipelining can improve performance. Figure 3.4.1B shows a two-stage pipelining scheme the I and E stages of two different instructions are performed simultaneously two stages of the pipeline are an instruction fetch stage, and an execute/memory two stage that executes the instruction, including register-to-memory a to-register operations. Thus we see that the instruction fetch stage instruction can e performed in parallel with the first part of the ex. stage. However, the execute/memory stage of the second instruction must be delayed until the first instruction clears the second stage of the pipeline. This scheme up to twice the execution rate of a serial scheme. Two problems prevent the maximum speed up from being achieved. First, we assume that a single-port memory is used and that only one memory access is possible per stage. This requires the insertion of a wait state in some instructions. Second, a branch instruction is sequential flow of execution. To accommodate this with minimum circuit instruction can be inserted into the instruction stream by the compiler or assemblier.

Pipelining can be improved further by permitting two memory access stage. This yields the sequence shown in Figure 3.1.1 Now, up to three instructions can be overlapped; and the improvement_is as much as a factor of 3. Again, branch Instructions cause the speedup to fall short of the maximum possible. Also data dependencies have an effect. If an instruction needs an operand that is altered



Load	rA ← M	I	E	D										
Load	rB ← M				I	E	D							
Add	rC ← rA + rB							I	E					
Store	M ← rC									I	E	D		
Branch X													I	E

(a) Sequential execution

Load	rA ← M	I	E	D							
Load	rB ← M		I		E	D					
Add	rC ← rA + rB			I		E					
Store	M ← rC					I	E	D			
Branch X							I		E		
NOOP									I	E	

(b) Two-stage pipelined timing

Load	rA ← M	I	E	D					
Load	rB ← M		I	E	D				
NOOP				I	E				
Add	rC ← rA + rB				I	E			
Store	M ← rC					I	E	D	
Branch	X						I	E	
NOOP								I	E

(c) Three-stage pipelined timing

Load	rA ← M	I	E_1	E_2	D							
Load	rB ← M		I	E_1	E_2	D						
NOOP				I	E_1	E_2						
NOOP					I	E_1	E_2					
Add	rC ← rA + rB					I	E_1	E_2				
Store	M ← rC						I	E_1	E_2	D		
Branch	X							I	E_1	E_2		
NOOP									I	E_1	E_2	
NOOP										I	E_1	E_2

(d) Four-stage pipelined timing

The Effects of Pipelining 3.1.3

by the preceding instruction, a delay is required. Again. this can be accomplished by a NOOP

The pipelining discussed so far works best if the three stages are of approximately equal duration. Because the E stage usually involves an ALU operation, it may be longer. In this case, we can divide into two substages:

- E_l: Register file read
- E_Z: ALU operation and register write

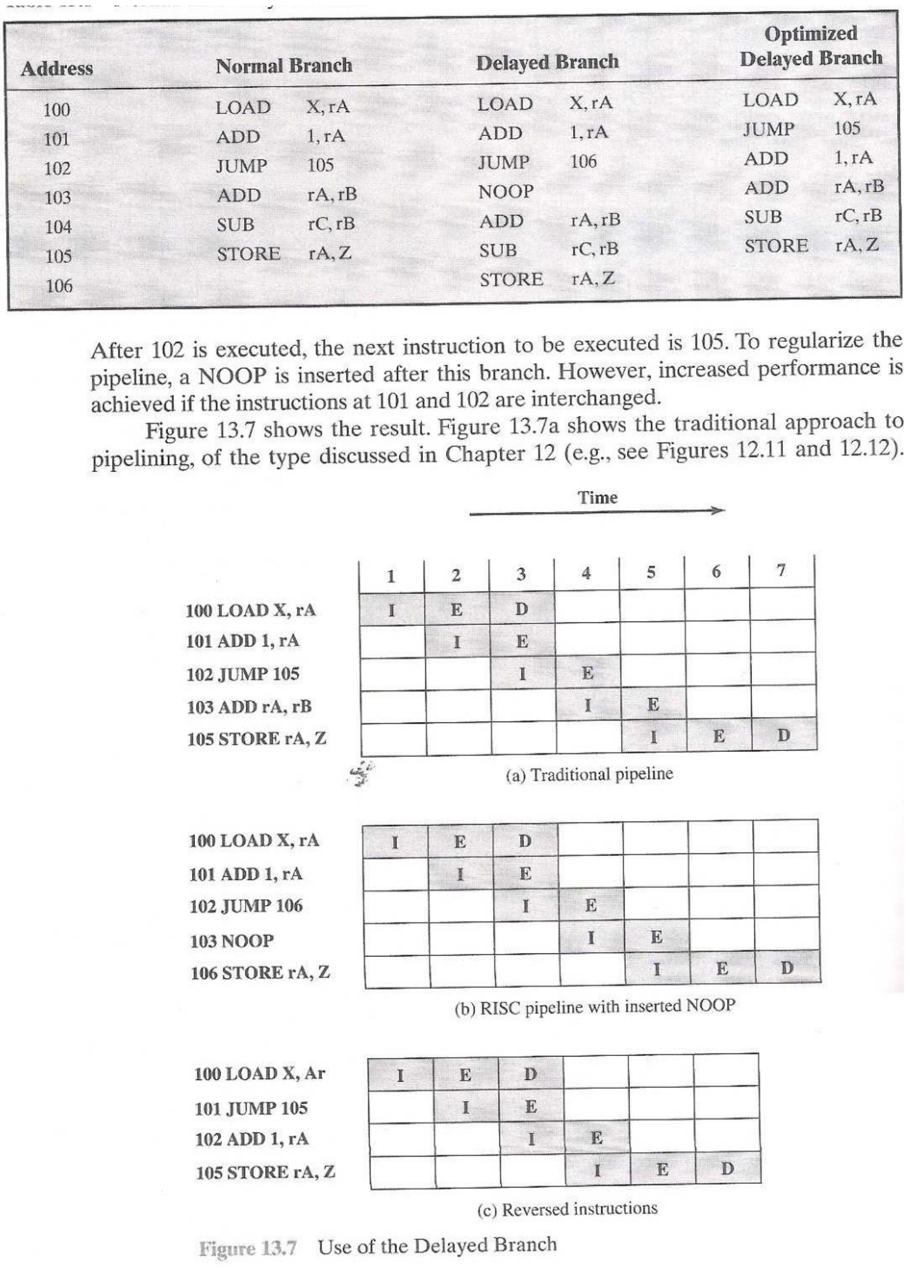
3.2 OPTIMIZATION OF PIPELINING

Because of the simplicity and regularity of a RISC instruction set, the design of the phasing into three or four stages is easily accomplished. Figure **3.1.3**shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

Because of the simple and regular nature of RISC instructions, pipelining schemes can be efficiently employed. There are few variations in instruction execution duration, and the pipeline can be tailored to reflect this. However, we have seen that data and branch dependencies reduce the overall execution rate.

Delayed Branch

To compensate for these dependencies, code reorganization techniques have been developed. First, let us consider branching instructions. ,belayed *branch*, a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after execution of the following instruction (hence the term *delayed).* The instruction location immediately following the branch is referred to the delay slot. *This* strange procedure is illustrated in Table 3.1.2 In the column labeled "normal branch," we see a normal symbolic instruction machine-language prow



Address	Normal Branch		Delayed Branch		Optimized Delayed Branch	
100	LOAD	X, rA	LOAD	X, rA	LOAD	X, rA
101	ADD	1, rA	ADD	1, rA	JUMP	105
102	JUMP	105	JUMP	106	ADD	1, rA
103	ADD	rA, rB	NOOP		ADD	rA, rB
104	SUB	rC, rB	ADD	rA, rB	SUB	rC, rB
105	STORE	rA, Z	SUB	rC, rB	STORE	rA, Z
106			STORE	rA, Z		

After 102 is executed, the next instruction to be executed is 105. To regularize the pipeline, a NOOP is inserted after this branch. However, increased performance is achieved if the instructions at 101 and 102 are interchanged.

Figure 13.7 shows the result. Figure 13.7a shows the traditional approach to pipelining, of the type discussed in Chapter 12 (e.g., see Figures 12.11 and 12.12).

Time →

	1	2	3	4	5	6	7
100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 105			I	E			
103 ADD rA, rB				I	E		
105 STORE rA, Z					I	E	D

(a) Traditional pipeline

100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

(b) RISC pipeline with inserted NOOP

100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

(c) Reversed instructions

Figure 13.7 Use of the Delayed Branch

The JUMP instruction is fetched at time 3. At time 4, the JUMP instruction is executed at the same time that instruction 103 (ADD instruction) is fetched. Because a JUMP occurs, which updates the program counter, the pipeline must be cleared of instruction 103; at time 5, instruction 105, which is the target of the JUMP, is loaded. Figure 3.1.1 shows the same pipeline handled by a typical RISC organization. The timing is the same. However, because of the insertion of the NOOP instruction, we do not need special circuitry to clear the pipeline; the HOOP simply executes with no effect. Figure 13.7c shows the use of the delayed branch. The JUMP instruction is fetched at time 2, before the ADD instruction, which is fetched at time 3. Note, however, that the ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. Therefore, during time 4, the ADD instruction is executed at the same time that instruction 105 is fetched. Thus, the original semantics of the program are retained but one less clock cycle is required for execution.

This interchange of instructions will work successfully for unconditional branches, calls, and returns. For conditional branches, this procedure cannot be blindly applied. If the condition that is tested for the branch can be altered by the immediately preceding instruction, then the compiler must refrain from doing the interchange and instead insert a NOOP Otherwise, the compiler can seek to insert a useful instruction after the branch. The experience with both the Berkeley RISC and IBM 801 systems is that the majority of conditional branch instructions can be optimized in this fashion ([PATT82a], [RADI83]).

Delayed load

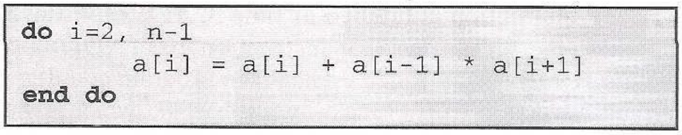
A similar sort of tactic, called the delayed load, can be used on LOAD instructions. On LOAD instructions, the register that is to be the target of the load is locked by the processor. The processor then continues execution of the instruction stream until it reaches an instruction requiring that register, at which point it idles until the load is complete. If the compiler can rearrange instructions so that useful work can be done while the load is in the pipeline, efficiency is increased.

Loop unrolling

Another compiler technique to improve instruction parallelism is loop unrolling [BAC094]. Unrolling replicates the body of a loop some number of times called the unrolling factor (u) and iterates by step a instead of step 1. Unrolling can improve the performance by

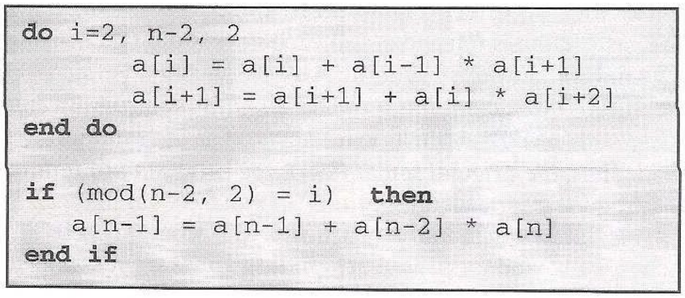
- reducing loop overhead
- increasing instruction parallelism by improving pipeline performance
- improving register, data cache, or TLB locality

Figure 13.8 illustrates all three of these improvements in an example. Loop overhead is cut in half because two iterations are performed before the test and



```
do i=2, n-1
        a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) Original loop



```
do i=2, n-2, 2
        a[i] = a[i] + a[i-1] * a[i+1]
        a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = i)  then
   a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) Loop unrolled twice

branch at the end of the loop. Instruction parallelism is increased because the second assignment can be performed while the results of the first are being stored and the loop variables are being updated. If array elements are assigned to registers, register locality will improve because a[i] and a[i + 1] are used twice in the loop body, reducing the number of loads per iteration from three to two.

As a final note, we should point out that the design of the instruction pipeline should not be carried out in isolation from other optimization techniques applied to the system. For example, [BRAD91b] shows that the scheduling of instructions for the pipeline and the dynamic allocation of registers should be considered together to achieve the greatest efficiency.

UNIT 4: MIPS R4000

One of the first commercially available RISC chip sets was developed by MIPS Technology Inc. The system was inspired by an experimental system, also using the name MIPS, developed at Stanford [HENN84]. In this section we look at the MIPS 84000. It has substantially the same architecture and instruction set of the earlier MIPS designs: the 82000 and 83000. The most significant difference is that the 84000 uses 64 rather than 32 bits for all internal and external data paths and for addresses, registers, and the ALU.

The use of 64 bits has a number of advantages over a 32-bit architecture. It allows a bigger address space-large enough for an operating system to map more than a terabyte of files directly into virtual memory for easy access. With 1-terabyte and larger disk drives now common, the 4-gigabyte address space of a 32-bit machine becomes limiting. Also, the 64-bit capacity allows the 84000 to process data such as IEEE double-precision floating-point numbers and character strings, up to eight characters in a single action.

The R4000 processor chip is partitioned into two sections, one containing the CPU and the other containing a coprocessor for memory management. The processor has a very simple architecture. The intent was to design a system in which the instruction execution logic was as simple as possible, leaving space available for logic to enhance performance (e.g., the entire memory-management unit).

The processor supports thirty-two 64-bit registers. It also provides for up to 128 Kbytes of high-speed cache, half each for instructions and data. The relatively large cache (the IBM 3090 provides 128 to 256 Kbytes of cache) enables the system to keep large sets of program code and data local to the processor, off-loading the main memory bus and avoiding the need for a large register file with the accompanying windowing logic.

3.1 INSTRUCTION SET

Table 13.9 lists the basic instruction set for all MIPS R series processors. All processor instructions are encoded in a single 32-bit word format. All data operations are register to register; the only memory references are pure load/store operations.

The R4000 makes no use of condition codes. If an instruction generates a condition, the corresponding flags are stored in a general-purpose register. This avoids the need for special logic to deal with condition codes as they affect the pipelining mechanism and the reordering of instructions by the compiler. Instead, the mechanisms already implemented to deal with register-value dependencies are employed. Further, conditions mapped onto the register files are subject to the same compile-time optimizations in allocation and reuse as other values stored in registers.

As with most RISC-based machines, the MIPS uses a single 32-bit instruction length. This single instruction length simplifies instruction fetch and decode, and it also simplifies the int action of instruction fetch with the virtual memory management unit (i.e., instructions do not cross word or page boundaries). The three instruction formats (Figure 13.9) share common formatting of opcodes and register references, simplifying instruction decode. The effect of more complex instructions can be synthesized at compile time.

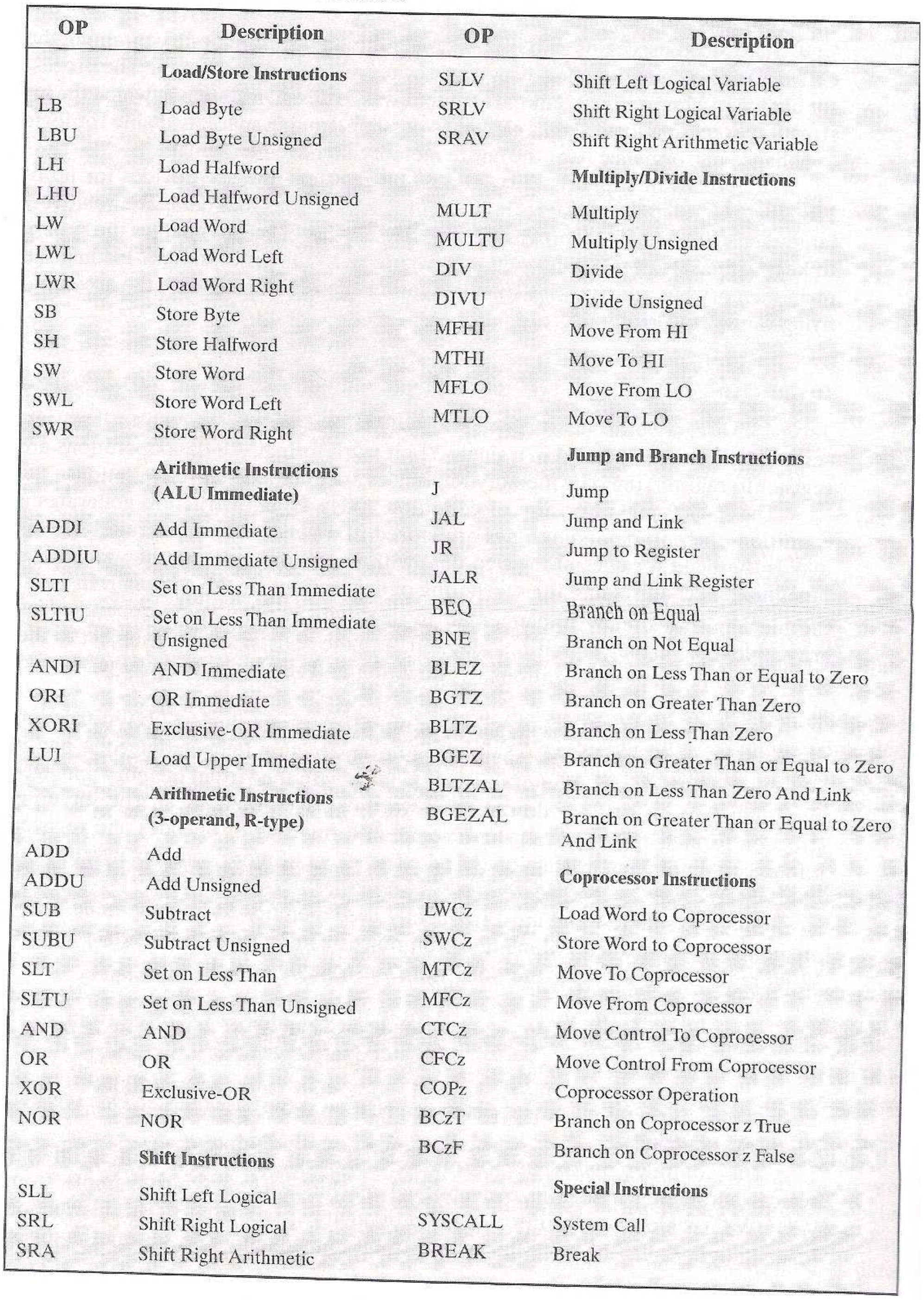
Only the simplest and most frequently used memory-addressing mode is implemented in hardware. All memory references consist of a 16-bit offset from a 32-bit register. For example, the "load word" instruction is of the form

1w r2, 128(r3) /"load word at address 128 offset from register 3 into register 2

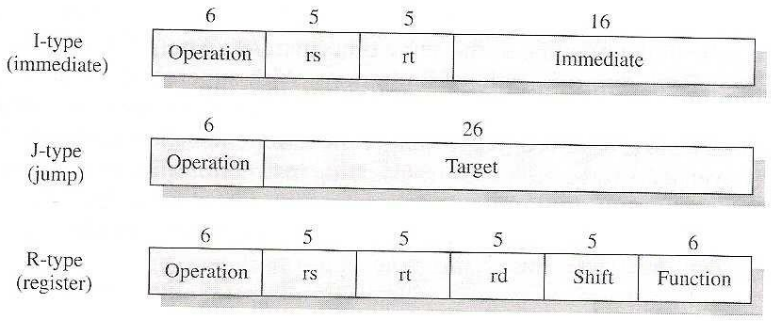
Each of the 32 general-purpose registers can be used as the base register. One r0, always contains 0.

The compiler makes use of multiple machine instructions to typical addressing modes in conventional machines. Here is an e [CHOW87], which uses the instruction lui (load upper immediate). loads the upper half of a register with a 16-bit immediate value, setting the lower

Table 13.9 MIPS R-Series Instruction Set



OP	Description	OP	Description
	Load/Store Instructions	SLLV	Shift Left Logical Variable
LB	Load Byte	SRLV	Shift Right Logical Variable
LBU	Load Byte Unsigned	SRAV	Shift Right Arithmetic Variable
LH	Load Halfword		**Multiply/Divide Instructions**
LHU	Load Halfword Unsigned	MULT	Multiply
LW	Load Word	MULTU	Multiply Unsigned
LWL	Load Word Left	DIV	Divide
LWR	Load Word Right	DIVU	Divide Unsigned
SB	Store Byte	MFHI	Move From HI
SH	Store Halfword	MTHI	Move To HI
SW	Store Word	MFLO	Move From LO
SWL	Store Word Left	MTLO	Move To LO
SWR	Store Word Right		**Jump and Branch Instructions**
	Arithmetic Instructions (ALU Immediate)	J	Jump
ADDI	Add Immediate	JAL	Jump and Link
ADDIU	Add Immediate Unsigned	JR	Jump to Register
SLTI	Set on Less Than Immediate	JALR	Jump and Link Register
SLTIU	Set on Less Than Immediate Unsigned	BEQ	Branch on Equal
		BNE	Branch on Not Equal
ANDI	AND Immediate	BLEZ	Branch on Less Than or Equal to Zero
ORI	OR Immediate	BGTZ	Branch on Greater Than Zero
XORI	Exclusive-OR Immediate	BLTZ	Branch on Less Than Zero
LUI	Load Upper Immediate	BGEZ	Branch on Greater Than or Equal to Zero
	Arithmetic Instructions (3-operand, R-type)	BLTZAL	Branch on Less Than Zero And Link
ADD	Add	BGEZAL	Branch on Greater Than or Equal to Zero And Link
ADDU	Add Unsigned		**Coprocessor Instructions**
SUB	Subtract	LWCz	Load Word to Coprocessor
SUBU	Subtract Unsigned	SWCz	Store Word to Coprocessor
SLT	Set on Less Than	MTCz	Move To Coprocessor
SLTU	Set on Less Than Unsigned	MFCz	Move From Coprocessor
AND	AND	CTCz	Move Control To Coprocessor
OR	OR	CFCz	Move Control From Coprocessor
XOR	Exclusive-OR	COPz	Coprocessor Operation
NOR	NOR	BCzT	Branch on Coprocessor z True
	Shift Instructions	BCzF	Branch on Coprocessor z False
SLL	Shift Left Logical		**Special Instructions**
SRL	Shift Right Logical	SYSCALL	System Call
SRA	Shift Right Arithmetic	BREAK	Break



Operation	Operation code
rs	Source register specifier
rt	Source/destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU/shift function specifier

Figure 13.9 MIPS Instruction Formats

half to zero. Consider an assembly-language instruction that uses a 32-bit immediate argument

lw r2, #imm (r4) /` load word at address using a 32-bit immediate offset #imm
/'* offset from register 4 into register 2

This instruction can be compiled into the following MIPS instructions

lui r~,, #imm-hi /* where #imm-hi is the high-order 16 bits of #imm
addu rl, rl, r4 /^{1}` add unsigned #imm-hi to r4 and put in r1
lw r2, #imm-lo (rl) /* where #imm-lo is the low-order 16 bits of #imm

3.2 INSTRUCTION PIPELINE

With its simplified instruction architecture, the MIPS can achieve very efficient pipelining. It is instructive to look at the evolution of the MIPS pipeline, as it illustrates the evolution of RISC pipelining in general.

The initial experimental RISC systems and the first generation of commercial RISC processors achieve execution speeds that approach one instruction per system clock cycle. To improve on this performance, two classes of processors have evolved ' t`a lv vxuccthon a&¯md MVte¯lnsttrudlons per CLOCK cydie: superscdlar ants superpipelined architectures. In essence, a superscalar architecture replicates each of the pipeline stages so that two or more instructions at the same stage of the pipeline can be processed simultaneously. A super pipelined architecture is one that makes use of more, and more fine-grained, pipeline stages. With more stages, more instructions can be in the pipeline at the same time, increasing parallelism.

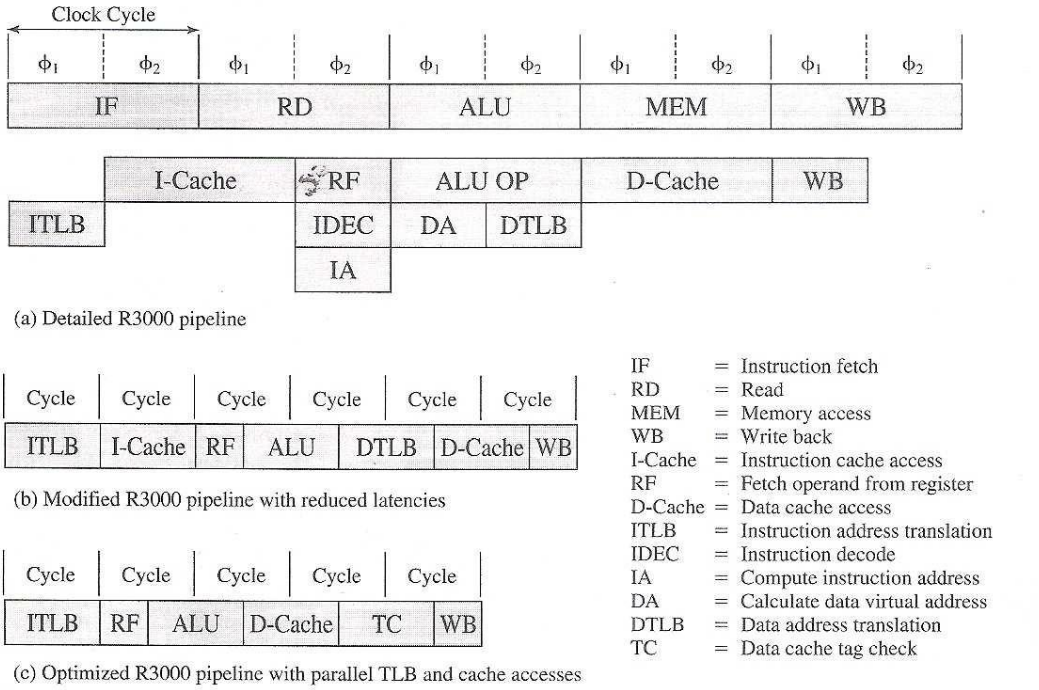
Both approaches have limitations. With superscalar pipelining, dependencies between instructions in different pipelines can slow down the system. Also, overhead logic is required to coordinate these dependencies. With superpipelining, there is overhead associated with transferring instructions from one stage to the next.

Chapter 14 is devoted to a study of superscalar architecture. The MIPS 84000 is a good example of a RISC-based superpipeline architecture.

MIPS R3000 Five-Stage Pipeline Simulator

Figure 13.10a shows the instruction pipeline of the 83000. In the 83000, the pipeline advances once per clock cycle. The MIPS compiler is able to reorder instructions to fill delay slots with code 70 to 90% of the time. All instructions follow the same sequence of five pipeline stages:

- Instruction fetch
- Source operand fetch from register.



(a) Detailed R3000 pipeline

(b) Modified R3000 pipeline with reduced latencies

(c) Optimized R3000 pipeline with parallel TLB and cache accesses

IF	=	Instruction fetch
RD	=	Read
MEM	=	Memory access
WB	=	Write back
I-Cache	=	Instruction cache access
RF	=	Fetch operand from register
D-Cache	=	Data cache access
ITLB	=	Instruction address translation
IDEC	=	Instruction decode
IA	=	Compute instruction address
DA	=	Calculate data virtual address
DTLB	=	Data address translation
TC	=	Data cache tag check

Figure 13.10 Enhancing the R3000 Pipeline

- ALU operation or data operand address generation

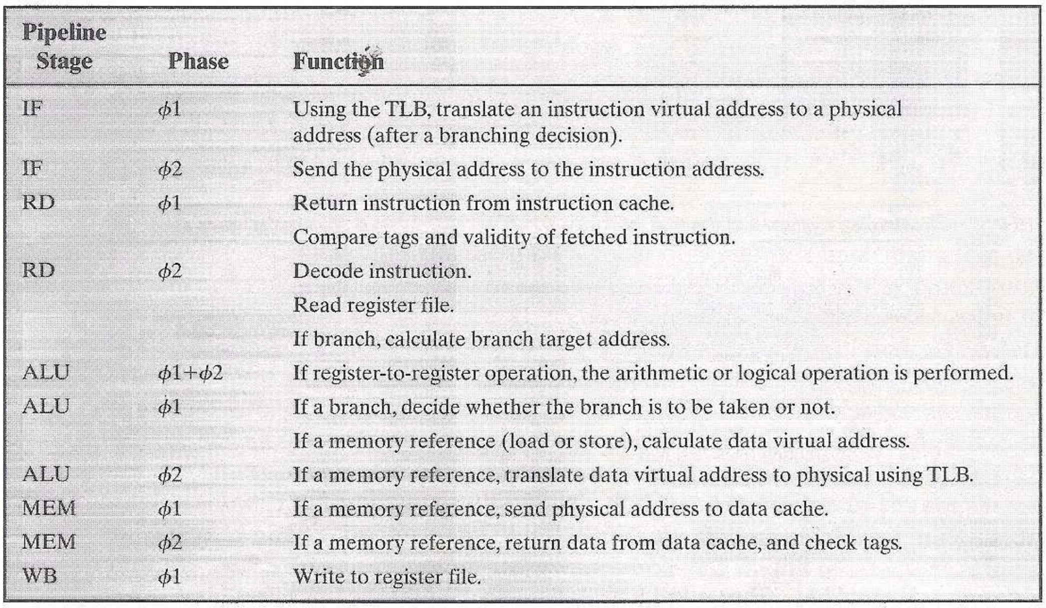
- Data memory reference
- Write back into register file

As illustrated in Figure 3.1.5a, there is not only parallelism due to pipelining but also parallelism within the execution of a single instruction. The 60-ns clock cycle is divided into two 30-ns stages. The external instruction and data access operations to the cache each require 60 ns, as do the major internal operations (OP, DA, IA). Instruction decode is a simpler operation, requiring only a single 30-ns stage, overlapped with register fetch in the same instruction. Calculation of an address for a branch instruction also overlaps instruction decode and register fetch, so that a branch at instruction i can address the ICACHE access of instruction i + 2. Similarly, a load at instruction i fetches data that are immediately used by the OP of instruction i + 1, while an ALU/shift result gets passed directly into instruction 1 with no delay. This tight coupling between instructions makes for a highly efficient pipeline.

In detail, then, each clock cycle is divided into separate stages, denoted as 01 and 02. The functions performed in each stage are summarized in Table 3.1.5a.

The 84000 incorporates a number of technical advances over the 83000. The use of more advanced technology allows the clock cycle time to be cut in half, to 30 ns, and for the access time to the register file to be cut in half. In addition, there is greater density on the chip, which enables the instruction and data caches to be incorporated on the chip. Before looking at the final 84000 pipeline, let us consider how the 83000 pipeline can be modified to improve performance using 84000 technology.

Figure 3.5.b shows a first step. Remember that the cycles in this figure are half as long as those in Figure 3.5.b. Because they are on the same chip, the instruction



Pipeline Stage	Phase	Function
IF	$\phi 1$	Using the TLB, translate an instruction virtual address to a physical address (after a branching decision).
IF	$\phi 2$	Send the physical address to the instruction address.
RD	$\phi 1$	Return instruction from instruction cache. Compare tags and validity of fetched instruction.
RD	$\phi 2$	Decode instruction. Read register file. If branch, calculate branch target address.
ALU	$\phi 1+\phi 2$	If register-to-register operation, the arithmetic or logical operation is performed.
ALU	$\phi 1$	If a branch, decide whether the branch is to be taken or not. If a memory reference (load or store), calculate data virtual address.
ALU	$\phi 2$	If a memory reference, translate data virtual address to physical using TLB.
MEM	$\phi 1$	If a memory reference, send physical address to data cache.
MEM	$\phi 2$	If a memory reference, return data from data cache, and check tags.
WB	$\phi 1$	Write to register file.

and data cache stages take only half as long; so they still occupy only one clock cycle. Again, because of the speedup of the register file access, register read and write still occupy only half of a clock cycle.

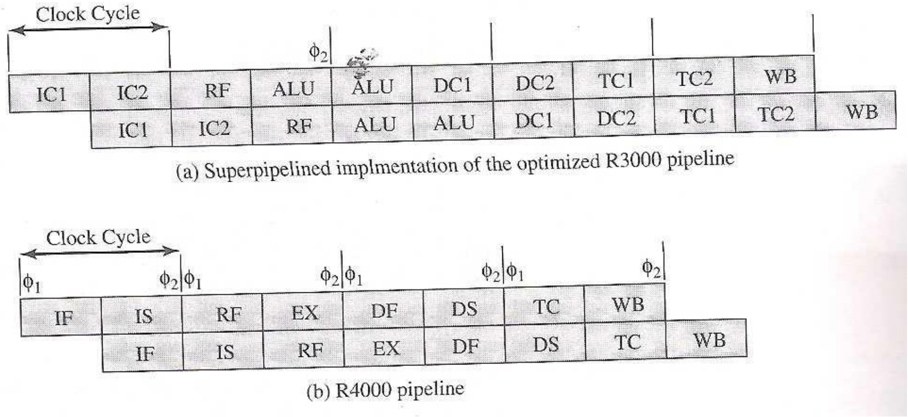
Because the R4000 caches are on-chip, the virtual-to-physical address translation can delay the cache access. This delay is reduced by implementing virtually indexed caches and going to a parallel cache access and address translation. Figure 3.1.5c shows the optimized R3000 pipeline with this improvement. Because of the compression of events, the data cache tag check is performed separately on the next cycle after cache access. This check determines whether the data item is in the cache.

In a super pipelined system, existing hardware is used several times per cycle by inserting pipeline registers to split up each pipe stage. Essentially, each super pipeline stage operates at a multiple of the base clock frequency, the multiple depending on the degree of super pipelining. Tke R4000 technology has the speed and density to permit super pipelining of degree 2. Figure 13.11 a shows the optimized R3000 pipeline using this super pipelining. Note that this is essentially the same dynamic structure as Figure 3.1.5c

Further improvements can be made. For the R4000, a much larger and specialized adder was designed. This makes it possible to execute ALU operations at twice the rate. Other improvements allow the execution of loads and stores at twice the rate.

The R4000 has eight pipeline stages, meaning that as many as eight instructions can be in the pipeline at the same time. The pipeline advances at the rate of two stages per clock cycle. The eight pipeline stages are as follows:

Instruction fetch first half: Virtual address is presented to the instruction cache and the translation lookaside buffer.



(a) Superpipelined implmentation of the optimized R3000 pipeline

(b) R4000 pipeline

Theoretical R3000 and Actual R4000 Superpipelines

IF = Instruction fetch first half DC = Data cache
IS = Instruction fetch second half DF = Data cache first half
RF = Fetch operands fromDS = Data cache second half
EX = Instruction execute TC = Tag check
IC = Instruction cache

- **Instruction fetch second half:** Instruction cache outputs the instruction and the TLB generates the physical address.
- **Register file:** Three activities occur in parallel:
  - Instruction is decoded and check made for interlock conditions (i.e., this instruction depends on the result of a preceding instruction).
  - Instruction cache tag check is made.
  - Operands are fetched from the register file.
  - **Instruction execute:** One of three activities can occur:
  - If the instruction is a register-to-register operation, the ALU performs the arithmetic or logical operation.
  - If the instruction is a load or store, the data virtual address is calculated.
  - If the instruction is a branch, the branch target virtual address is calculated and branch conditions are checked.
  - Data cache first: Virtual address is presented to the data cache and TLB.
  - **Data cache second:** The TLB generates the physical address, and the data cache outputs the instruction.
  - **Tag check:** Cache tag checks are performed for loads and stores.

- **Write back:** Instruction result is written back to register file.

4.0 CONCLUSION

This unit has motivated the key characteristics of RISC machines:

1. A limited instruction set with a fixed format
2. A large number of registers or the use of a compiler that optimizes register usage and
3. An emphasis optimizing the instruction pipeline

5.0 SUMMARY

A RISC instruction set architecture also lends itself to the delayed branch technique, in which branch instructions are rearranged with other instruction to improve pipeline efficiency.

6.0 TUTOR MARKED ASSIGNMENT

1. Briefly explain the two basic approaches used to minimize register- memory operations on RISC machines.
2. List the advantages of a R4000 of 64 bits over a 32 bit architecture.

7.0 REFERENCES/ FURTHER READING

Kane G and Heinrich . TMIPS RISC Architecture Engle wood Cliffs NJ Prentice Hall, 1992.

MODULE 5: OPERATING SYSTEM SUPPORT ERROR DETECTION AND ERROR CORRECTION CODING

UNIT 1: OPERATING SYSTEM OVERVIEW
UNIT 2: SCHEDULING
UNIT 3: MEMORY SYSTEM

1.0 INTRODUCTION

The operating system (OS) is the soft ware that controls the execution of programs on a processor and that manages the processors resources.

2.0 OBJECTIVES

At the end of this unit, you should be able to

- Understand the meaning of operating system
- Explain the functions of operating system
- Discuss the operating system objectives

3.1 Operating System Objectives And Function

An OS is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware. It can be thought of as having two objectives:

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.

Let us examine these two aspects of an OS in turn.

The operating system a as user/ computer interface

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure 8.1. The user of those applications, the end user, generally is not concerned with the computer's architecture. Thus the end user views a computer system in terms of an application. That application row can used in a programming language and is developed by an application programme. To develop an application program as a set of processor instructions that is completely responsible for controlling the computer hardware would be an overwhelmingly complex task. To ease this task, a set of systems programs is provided. Some of these programs are referred to as **utilities.** These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer makes use of these facilities in developing an application, and the application, while it is running, invokes the utilities to perform certain functions. The most important system program is the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

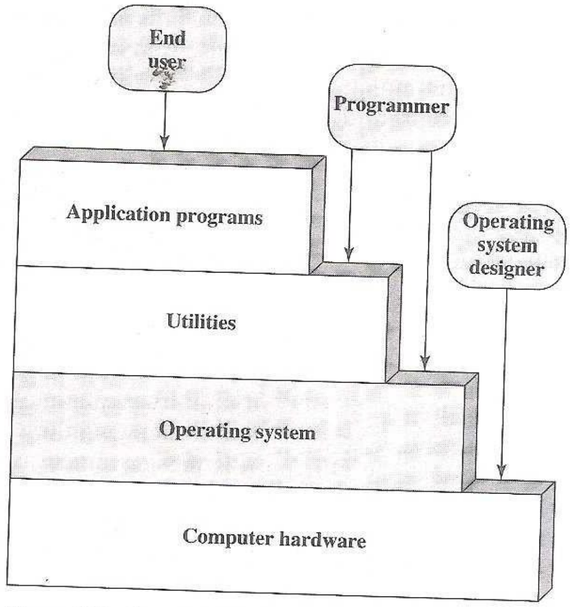


Figure 8.1 Layers and Views of a Computer System

Briefly. the OS typically provides services in the following area:

- ✓ **Program creation:** The OS provides a variety of facilities and services such as editors and debuggers to assist the programmer in creating programs. Typically these services are in the form of utility programs that are part of the OS but are accessible through the OS.**Program execution:** A number of tasks need to be performed to program. Instructions and data must be loaded into main memory and files must be initialized and other resources must be prepared handles all of this for the user.
- ✓ **Access to I/O devices**: Each I/O device requires its own specifications or control signals for operation. The OS takes care of the details programmer can think in terms of simple reads and writes.
- ✓ **Controlled access to files:** In the case of files control must include an understanding of not only the nature of the I/O device (disk drive, tap also the file format on the storage medium. Again the OS worrit ndetails Further in the case of a system with multiple simultaneous OS can provide protection mechanisms to control access to the files.
- ✓ **System access:** In the case of a shared or public system, the OS controls access the system as a Nyhole and to specific system resources. The access must provide protection of resources and data from unauthorized must resolve conflicts for resource contention.

- ✓ **Error detection and response:** A variety of errors can occur while system is running. These include internal and external hardware errors such as a memory error or a device failure or malfunction; and various software errors such as arithmetic overflow attempt to access forbidden memory location and inability of the OS to grant the request of an application. In each case the OS must make the response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error to retrying the operation, to simply reporting the error to the application.
- ✓ **Accounting:** A good OS collects usage statistics for various resource monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancement tuning the system to improve performance. On a multiuser system the information can be used for billing purposes.

The operating system as resource manager

A computer is resources for the movement, storage, and processing of data and for theca these functions. The OS is responsible for managing these resources.

Can we say that the OS controls the movement, storage, and process data? From one point of view, the answer is yes: By managing the computer’s resources, the OS is in control of the computer's basic functions. But this control is exercised in a curious way. Normally, we think of a control mechanism as some external to that which is controlled, or at least as something that is a distinct separate part of that which is controlled. (For example, a residential heating is controlled by a thermostat, which is completely distinct from the heat-generation and heat-distribution apparatus.) This is not the case with the OS, which as a control mechanism is unusual in two respects:

The OS functions in the same way as ordinary computer software; that is, it is a program executed by the processor.

The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

The OS is, in fact, nothing more than a computer program. Like other computer programs, it provides instructions for the processor. The key difference is in the intent of the program. The OS directs the processor in the use of the other system

resources and in the timing of its execution of other programs. But in order for the processor to do any of these things, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some "useful" work and then resumes control long enough to prepare the processor to do the next piece of work. The mechanisms involved in all this should become clear as the chapter proceeds.

Figure 3.1.2 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the kernel, or nucleus, which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the OS and memory management hardware in the processor, as we shall see.

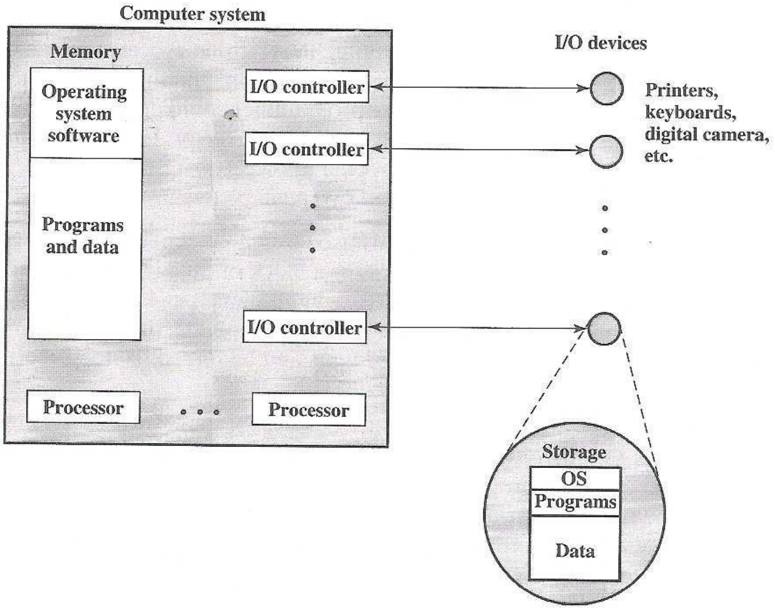


Figure 8.2 The Operating System as Resource Manager

3.2 Types of operating system

Certain key characteristics serve to differentiate various types of operating systems. The characteristics fall along two independent dimensions. The first dimension specifies whether the system is batch or interactive. In an **interactive** system, the user/ programmer interacts directly with the computer, usually through a keyboard/display terminal, to request the execution of a job or to perform a transaction. Furthermore, the user may, depending on the nature of the application, communicate with the computer during the execution of the job. A batch system is the opposite of interactive. The user's program is batched together with programs from other users and submitted by a computer operator. After the program is completed, results are printed out for the user. Pure batch systems are rare today. However, it will be useful to the description of contemporary operating systems to examine batch systems briefly.

An independent dimension specifies whether the system employs **multiprogramming** or not. With multiprogramming, the attempt is made to keep the processor as busy as possible, by having it work on more than one program at a time. Several programs are loaded into memory, and the processor switches rapidly among them. The alternative is a uniprogramming system that works only one program at a time.

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These processors were run from a console, consisting of display lights, toggle switches, some form of input device, and a printer. Programs in processor code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. The programmer could proceed to examine registers and main memory to determine the cause of the error. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems.:

Scheduling: Most installations used a sign-up sheet to reserve processor time. Typically, a user could sign up for a block of time in multiples of a half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer idle time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.

Setup time: A single program, called a job, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program), and then loading and linking together the object program and common functions. Each of these steps could in mounting or dismounting tapes, or setting up card decks. If an error occur the hapless user typically had to go back- to the beginning of the sequence. Thus a considerable amount of time was spent just in setting of program to run.

This mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and 1/O driver routines that were available as common software for all users.

Early processors were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, simple batch operating systems were developed. With such a system, also called a monitor, the user no longer has direct access to the processor. Rather, the user submits the job on cards or tape to a computer operator, who *batches* the jobs together sequentially and places the entire batch on an input device, for use by the monitor.

To understand how this scheme works, let us look at it from two points of view: that of the monitor and that of the processor. From the point of view of the monitor, the monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (Figure 3.1.2). That portion is referred to as the resident monitor. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are printed out for delivery to the user.

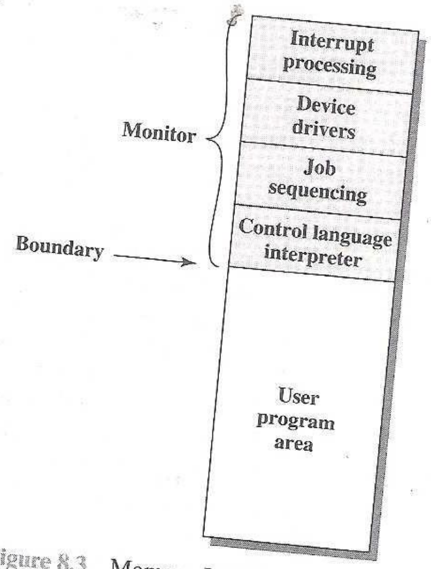


Figure 8.3 Memory Layout for a Resident Monitor

Now consider this sequence from the point of view of the processor. At a certain point in time, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read in to another portion of main memory. Once a job has been read in, the processor will encounter in the monitor a branch instruction that instructs the processor to continue execution at the start of the user program. The processor will then execute the instruction in the user's program until it encounters an ending or error condition. Either event causes the processor to fetch its next instruction from the monitor program. Thus the phrase "control is passed to a job" simply means that the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means that the processor is now fetching and executing instructions from the monitor program.

It should be clear that the monitor handles the scheduling problem. A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time.

How about the job setup time? The monitor handles this as well. With each job, instructions are included in a job control language (JCL). This is a special type of programming language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in FORTRAN plus some data to be used by the program. Each FORTRAN instruction and each item of data is on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning "$". The overall format of the job looks like this:

To execute this job, the monitor reads the $FTN line and loads the appropriate compiler from its mass storage (usually tape). The compiler translates the user's program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is referred to as "compile, load, and go." If it is stored on tape, then the $LOAD instruction is required. This instruction is read by the monitor, which regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory in place of the compiler and transfers control to it. In this manner, a large segment of main memory can be shared among different subsystems, although only one such subsystem could be resident and executing at a time.

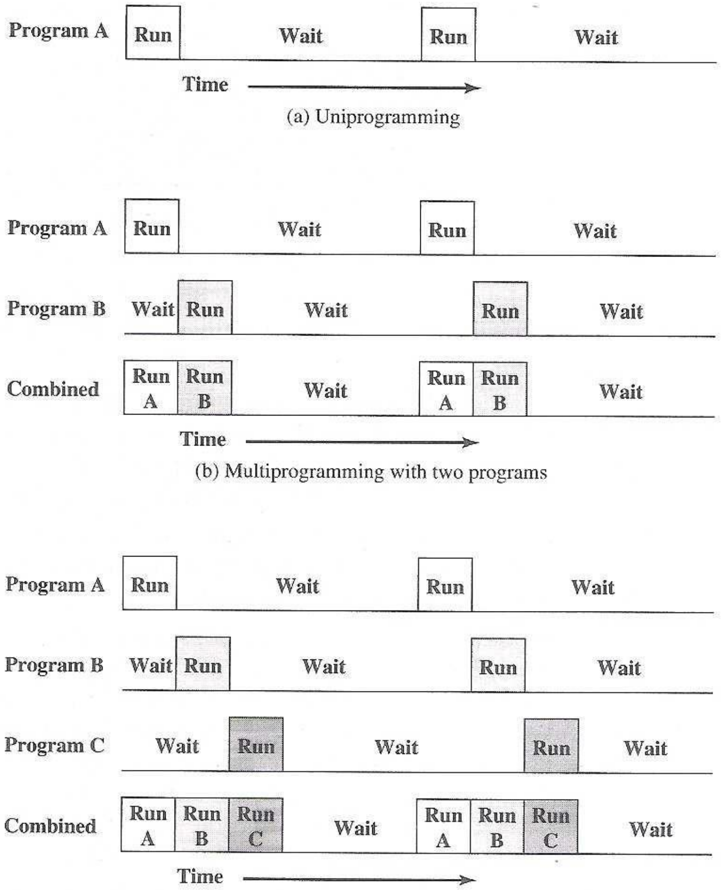
We see that the monitor, or batch OS, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main memory in order to seize and relinquish control alternately. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor. The monitor Would then abort the job. print out an error message. and load the next job.

- **Timer:** A timer is used to prevent a single job from monopolizing the system. The timer is set at the beginning of each job. If the timer expires an interrupt occurs. and control returns to the monitor.
- **Privileged instructions:** Certain instructions are designated privileged and can be executed only_ by the monitor. If the processor encounters such an instruction while executing a user program an error interrupt occurs. Among the privileged instructions are I/O instructions so that the monitor retains control of all I/O devices. This prevents. for example. a user program from accidentally reading job control instructions from the next job. If a user program wishes to perform I/O, it must request that the monitor perform the operation for it. If a privileged instruction is encountered by the processor while it is executing a user program, the processor hardware considers this an error and transfers control to the monitor.
- **Interrupts:** Early computer models did not have this capability. This feature gives the OS more flexibility in relinquishing control to and retraining control from user programs.

Processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Even with this overhead, the simple batch system improves utilization of the computer.

Even with the automatic job sequencing provided by a simple batch OS. The processor is often idle. The problem is that 1/O devices are slow compared to the processor. The calculation concerns a program that processes a file of records and performs, on average. 100 processor instructions per record. In this example the computer spends over 96% of its time waiting for I/O devices to finish transferring data! The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.



(a) Uniprogramming

(b) Multiprogramming with two programs

(c) Multiprogramming with three programs

Figure 8.5 Multiprogramming Example

This inefficiency is not necessary. We know that there must be enough memory to hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two user programs. Now, when one job needs to I/O. the processor can switch to the other job, which likely is not waiting for(Figure 8.5b). Furthermore we might expand memory to hold three, four programs and switch among all of them (Figure 8.Sc). This technique known as multiprogramming or multitasking It is the central theme of modern operating systems.

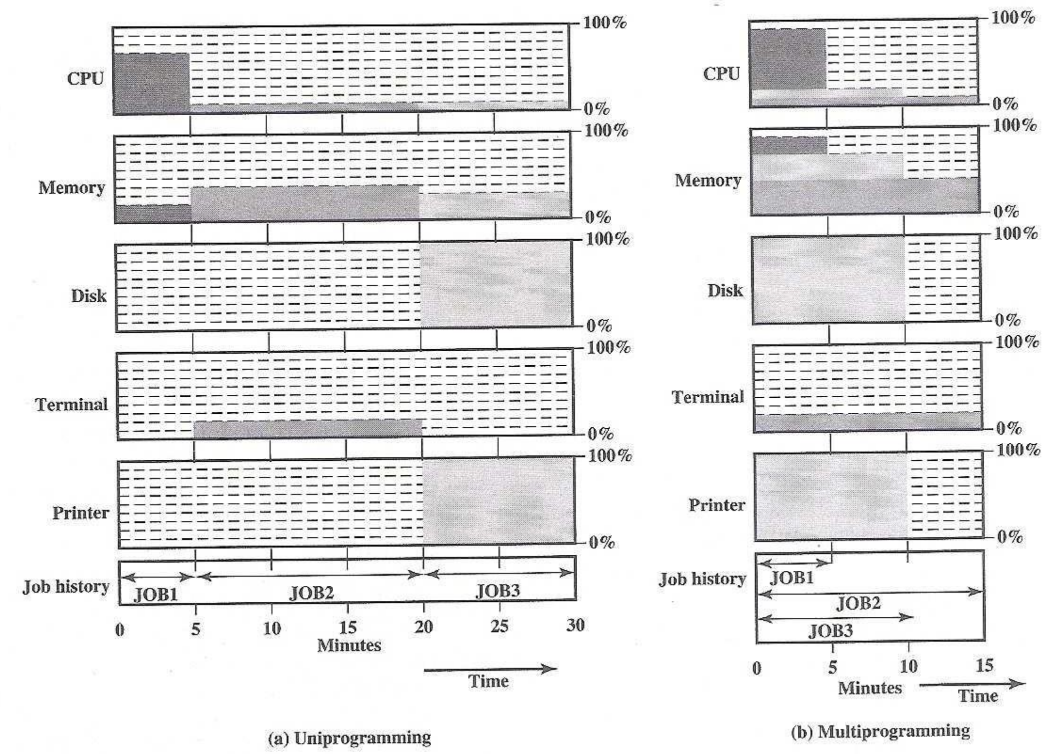


Figure 8.6 Utilization Histograms

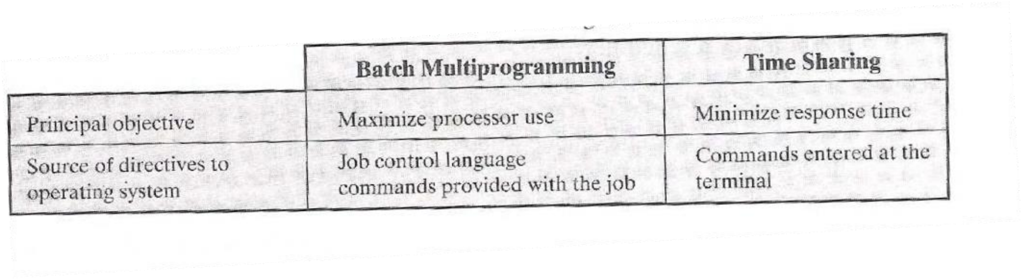
With interrupt-driven 1/O or DMA, the processor can issue an 1/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to^{w} an interrupt-handling program in the OS. The OS will then pass control to another job.

Multiprogramming operating systems are fairly sophisticated compared to singleprogram, or uniprogramming, systems. To have several jobs ready to run, the jobs must be kept in main memory, requiring some form of memory management. In addition, if several jobs are ready to run, the processor must decide which one to run, which requires some algorithm for scheduling. These concepts are discussed later in this chapter.

With the use of multiprogramming, batch processing can
be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated microcomputer. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because the processor's time is shared among multiple users. In a time-sharing system, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are n users actively requesting service at one time, each user will only see on the average 11n of the effective computer speed, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be comparable to that on a dedicated computer.



	Batch Multiprogramming	**Time Sharing**
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Both batch multiprogramming and time sharing use multiprogramming. The key differences are listed in Table 3.1.3

4.0 CONCLUSION

Operating system performs a number of functions namely; process scheduling and memory management can only be performs efficiently and rapidly if the processor hardware includes capabilities to support the operating system.

5.0 SUMMARY

The operating system determines which process should run at any given time. Typically the hardware will interrupt a running process from time to time to enables the operating system to make a new scheduling decision so as to share processor time fairly among a number of process.

The operating System is a program that manages the computers resources provides services for programmer and schedule the execution of the other programs

6.0 Tutor marked assignment

1. What is an operating system?
2. List and briefly define the key services provided by an operating system

7.0 References/ Further reading

Stallings W. Operating systems, internals design principles, sixth edition

UNIT 2: SCHEDULING

1.0 INTRODUCTION
2.0 OBJECTIVES
3.0 MAIN CONTENT
    3.1 LONG TERM SCHEDULING
    3.2 MEDIUM TERM SCHEDULING
    3.3 SHORT TERM SCHEDULING
4.0 CONCLUSION
5.0 SUMMARY
6.0 TUTOR MARKED ASSIGNMENT
7.0 REFERENCES/ FURTHER READING

1.0 INTRODUCTION

Scheduling is defined as a program in execution. It is the key to multi programming.

1.0 OBJECTIVES

At this end of this unit you should be able to

- Explain scheduling
- List and discuss types of scheduling

3.1 THE LONG TERM SCHEDULING

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming (number of processes in memory). Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

In a batch system, or for the batch portion of a general-purpose OS, newly submitted jobs are routed to disk and held in a batch queue. The long-term scheduler creates processes from the queue when it can. There are two decisions involved here. First, the scheduler must decide that the OS can take on one or more additional processes. Second, the scheduler must decide which job or jobs to accept and turn into processes. The criteria used may include priority, expected execution time, and I/O requirements.

For interactive programs in a time-sharing system, a process request is generated when a user attempts to connect to the system. Time-sharing users are not simply queued up and kept waiting until the system can accept them. Rather, the OS will accept all authorized comers until the system is saturated, using some predefined measure of saturation. At that point, a connection request is met with a message indicating that the system is full and the user should try again later.

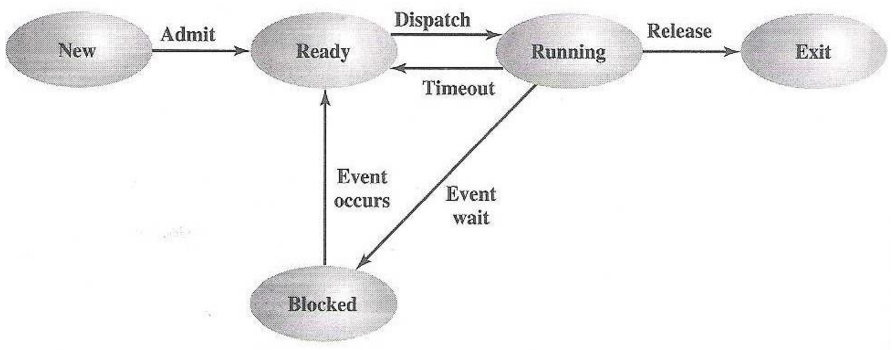
3.2 Medium-term scheduling

Medium-term scheduling is part of the swapping function, described in Section 8.3. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

3.3 Short-term scheduling

The long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process, and which one to take. The short-term scheduler, also known as the dispatcher, executes frequently and makes the fine-grained decision of which job to execute next.

To understand the operation of the short-term scheduler, we need to consider the concept of a process state. During the lifetime of a process, its status will change a number of times. Its status at any point in time is referred to as a *state*. The term *state* is used because it connotes that certain information exists that defines the status at that point. At minimum, there are five defined states for a process (Figure 3.1.1):



- **New:** A program is admitted by the high-level scheduler but is not yet ready to execute. The OS will initialize the process. moving it to the ready state.

Ready: The process is ready to execute and is awaiting access to the processor.

Running: The process is being executed by the processor.

Waiting: The process is suspended from execution waiting for some system resource such as FO.

Halted: The process has terminated and will be destroyed by the OS.

For each process in the system the OS must maintain information indicating the state of the process and other information necessary for process execution. For this purpose each process is represented in the OS by a process control **block** (Figure 3.12), which typically contains

Identifier: Each current process has a unique identifier.

State: The current state of the process (new. ready. and so on).

Priority: Relative priority level.

Program counter: The address of the next instruction in the program to be executed.

Memory pointers: The starting and ending locations of the process in memory.

Context data: These are data that are present in registers in the processor while the process is executing. For now, it is

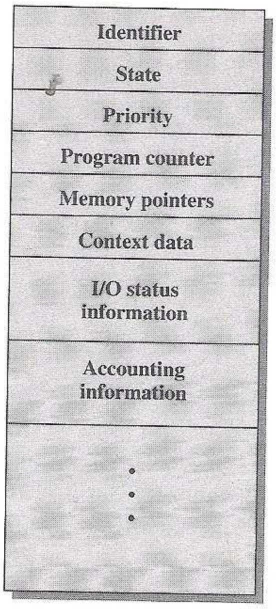


Figure 8.8 Process Control Block

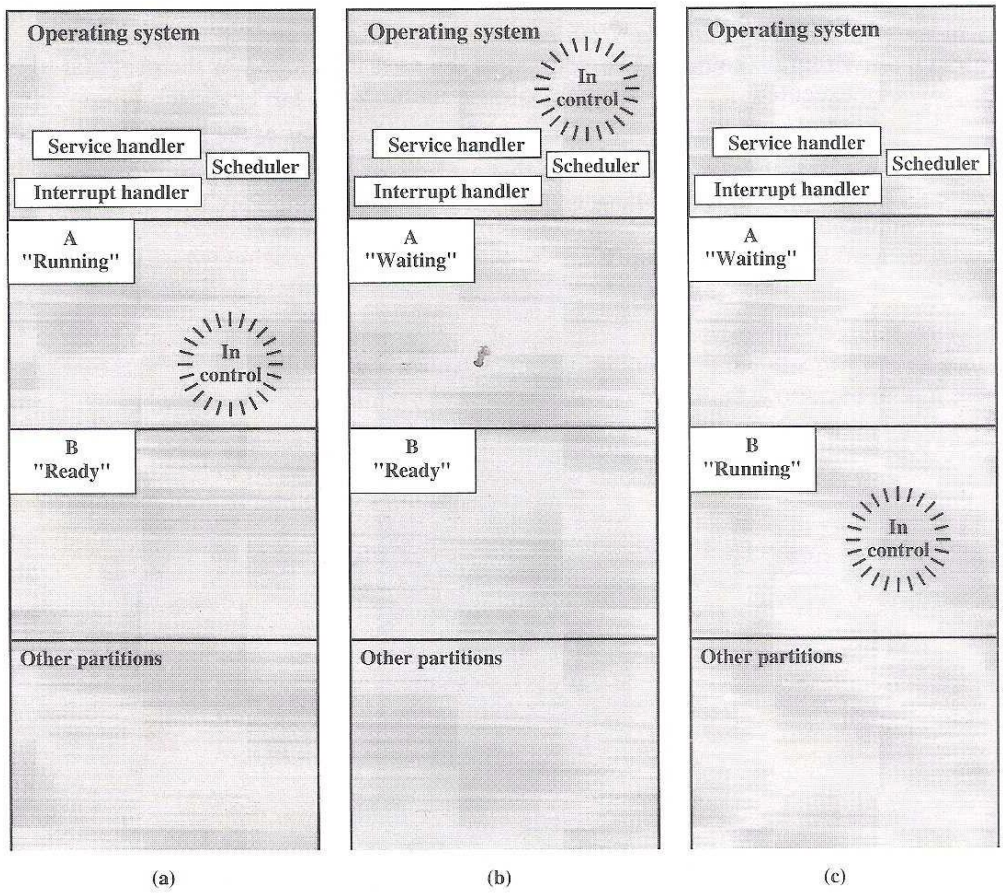
enough to say that these data represent the "context" of the process. The context data plus the program counter are saved when the process leaves the running state. They are retrieved by the processor when it resumes execution of the process.

I/O status information: Includes outstanding I/O requests, 1/0 devices (e.g., tape drives) assigned to this process, a list of files assigned to the process, and so on.

Accounting information: May include the amount of processor time and clock time used, time limits, account numbers, and so on.

When the scheduler accepts a new job or user request for execution, it creates a blank process control block and places the associated process in the new state. After the system has properly filled in the process control block, the process is transferred to the ready state.

To understand how the OS manages the scheduling of the various jobs in memory, let us begin by considering the simple example in Figure 8.9. The figure shows how main memory is partitioned at a given point in time. The kernel



of the OS is, of course, always resident. In addition, there are a number of active processes, including A and B, each of which is allocated a portion of memory.

We begin at a point in time when process A is running. The processor is executing instructions from the program contained in A's memory partition. At some later point in time, the processor ceases to execute instructions in A and begins executing instructions in the OS area. This will happen for one of three reasons:

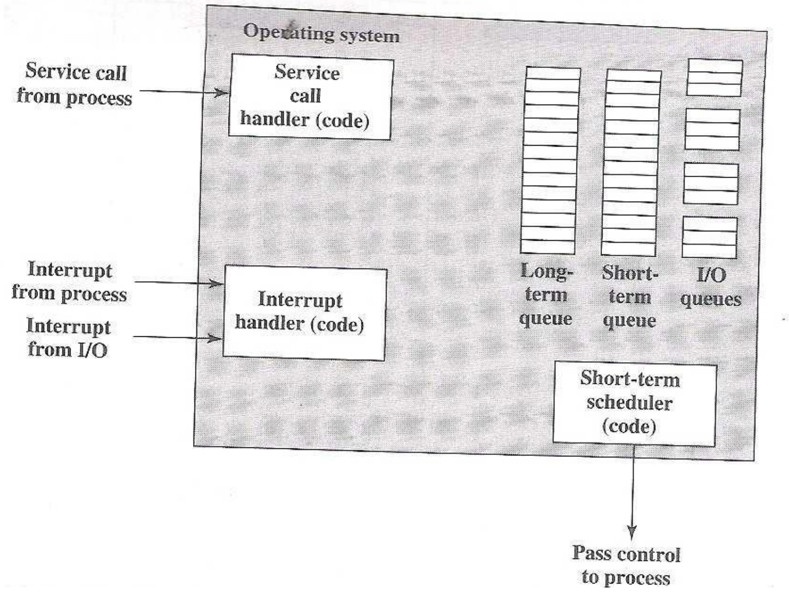
Process A issues a service call (e.g., an 1/O request) to the OS. Execution of A is suspended until this call is satisfied by the OS.

Process A causes an *interrupt.* An interrupt is a hardware-generated signal to the processor. When this signal is detected, the processor ceases to execute A and transfers to the interrupt handler in the OS. A variety of events related to A will cause an interrupt. One example is an error, such as attempting to execute a privileged instruction. Another example is a timeout; to prevent any one process from monopolizing the processor, each process is only granted the processor for a short period at a time.

Some event unrelated to process A that requires attention causes an interrupt. An example is the completion of an 1/O operation.

In any case, the result is the following. The processor saves the current context data and the program counter for A in As process control block and then begins executing in the OS. The OS may perform some work, such as initiating an I/O operation. Then the short-term-scheduler portion of the OS decides which process should be executed next. In this example, B is chosen. The OS instructs the processor to restore B's context data and proceed with the execution of B where it left off.

This simple example highlights the basic functioning of the short-term scheduler. Figure 5.10 shows the major elements of the OS involved in the multiprogramming and scheduling of processes. The OS receives control of the processor at the interrupt handler if an interrupt occurs and at the service-call handler if a service call occurs. Once the interrupt or service call is handled, the short-term scheduler is invoked to select a process for execution.



To do its job, the OS maintains a number of queues. Each queue is simply a waiting list of processes waiting for some resource. The long-term queue is a list of jobs waiting to use the system. As conditions permit, the high-level scheduler will allocate memory and create a process for one of the waiting items. The short-term queue consists of all processes in the ready state. Any one of these processes could use the processor next. It is up to the short-term scheduler to pick one. Generally, this is done with a round-robin algorithm, giving each process some time in turn. Priority levels may also be used. Finally, there is an I/O queue for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue.

Figure 8.11 suggests how processes progress through the computer under the control of the OS. Each process request (batch job, user-defined interactive job) is placed in the long-term queue. As resources become available, a process request becomes a process and is then placed in the ready state and put in the short-term queue. The processor alternates between executing OS instructions and executing user processes. While the OS is in control, it decides which process in the short-term queue should be executed next. When the OS has finished its immediate tasks, it turns the processor over to the chosen process.

As was mentioned earlier, a process being executed may be suspended for a variety of reasons. If it is suspended because the process requests I/O, then it is placed in the appropriate I/O queue. If it is suspended because of a timeout or because the OS must attend to pressing business. then it is placed in the ready state and put into the short-term queue.

Finally, we mention that the OS also manages the I/O queues. When an I/O operation is completed, the OS removes the satisfied process from that I/O queue and places it in the short-term queue. It then selects another waiting process (if any) and signals for the I/O device to satisfy that process's request.

UNIT 3: Memory System

1.0 Introduction

2.0 Objectives

3.0 Main content

3.1 Characteristics of memory systems

3.2 The memory hierarchy

3.3 Error correction

1.0 Introduction

Computer memory is organized into a hierarchy. At the highest level (closest to the processor) are the processor registers. Next comes one or more levels of cache, When multiple levels are used, they are denoted L1, L2, and so on.Error correction techniques are commonly used in memory systems.

2.0 objectives

At the end of this unit, you should be able to

-Understand and

- As one goes down the memory hierarchy, one finds decreasing cost/bit, increasing capacity, and slower access time. It would be nice to use only the fastest memory, but because that is the most expensive memory, we trade off access time for cost by using more of the slower memory. The design challenge is to organize th data and programs in memory so that the accessed memory words are usu~iII\ in the faster memory.
- In general, it is likely that most future accesses to train memory by the processor will be to locations recently accessed. So the cache automatically retains a copy of some of the recently used words from the DRAM. If the c~ichc is designed properly, then most of the time the processor will request memory words that are already in the cache.

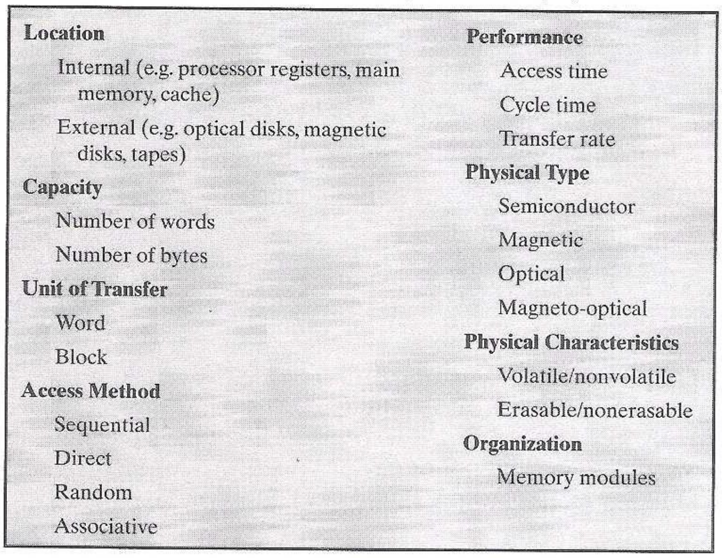
Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No one technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

This chapter and the next focus on internal memory elements, while Chapter 6 is devoted to external memory. To begin, the first section examines key characteristics of computer memories. The remainder of the chapter examines an essential element of all modern computer systems: cache memory.

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.

The term location in refers to whether memory is internal and external to the computer. Internal memory is often equated with main memory. But there are other forms of internal memory. The processor requires its own local memory, in

Table 4.1 Key Characteristics of Computer Memory Systems



Location
Internal (e.g. processor registers, main memory, cache)
External (e.g. optical disks, magnetic disks, tapes)
Capacity
Number of words
Number of bytes
Unit of Transfer
Word
Block
Access Method
Sequential
Direct
Random
Associative
Performance
Access time
Cycle time
Transfer rate
Physical Type
Semiconductor
Magnetic
Optical
Magneto-optical
Physical Characteristics
Volatile/nonvolatile
Erasable/nonerasable
Organization
Memory modules

the form of registers (e.g., see Figure 2.3). Further, as we shall see, the control unit portion of the processor may also require its own internal memory. We will defer discussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its capacity. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

A related incept is the unit of transfer. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64,128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

- **Word:** The "natural" unit of organization of memory. The size of the word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the **CRAY** C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.
- **Addressing units:** many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.
- **Unit of transfer:** For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable

unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

- **Sequential access:** Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in Chapter 6, are sequential access.
- **Direct access:** As with sequential access, direct access involves a shared read-write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in Chapter 6, are direct access.
- **Random access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independa t of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and performance. Three performance parameters are used:

- **Access time (latency):** For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read-write mechanism at the desired location.

- **Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.
- **Transfer rate:** This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time).

For non-random-access memory, the following relationship holds:

$T_x = T_A + R$ *(4.1)*

where

T_N = Average time to read or write N bits

T_A = Average access¯time

n = Number of bits

R = Transfer rate, in bits per second (bps)

A variety of physical types of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk a tape, and optical and magneto-optical.

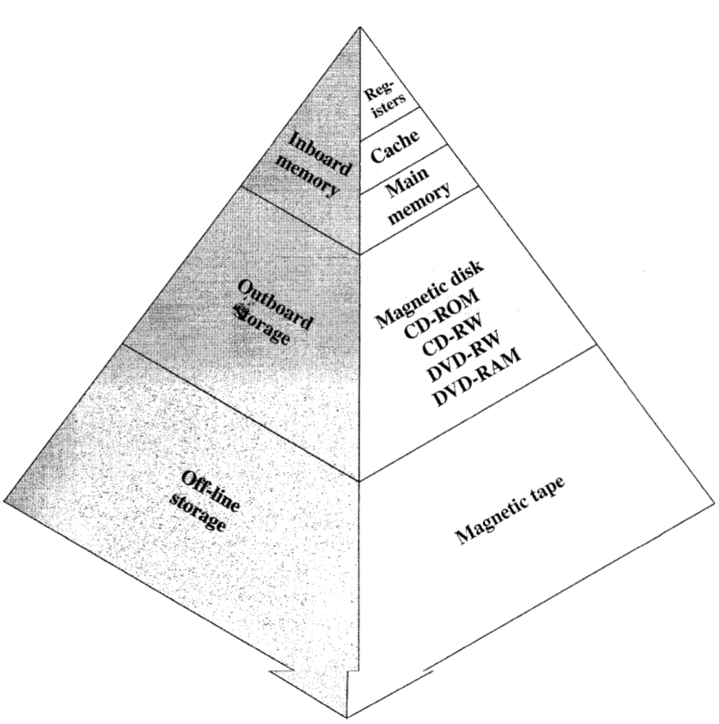
Several physical characteristics of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deteriorate until deliberately changed; no electrical power is needed to retain information .Magnetic-surface memories are nonvolatile. Semiconductor memory may be ¯volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroy the storage unit. Semiconductor memory of this type is known as *read- only mere* (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

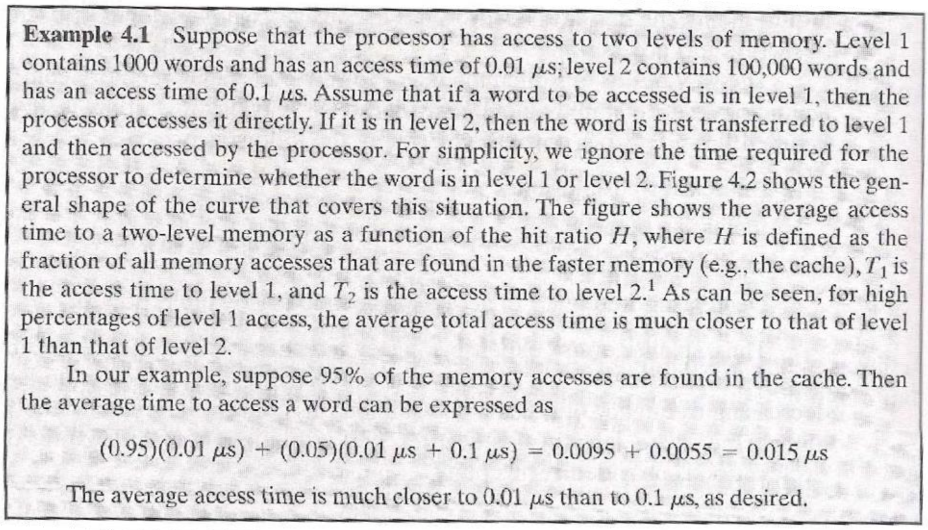
For random-access memory, the organization is a key design issue. By on *nation is* meant the physical arrangement of bits to form words. The arrangement is not always used, as is explained in Chapter 5.

The design constraints on a computer's memory can be summed up by three question:
How much? How fast? How expensive?
The question of how much is somewhat open ended. If the capacity is
applications will likely be developed to use it. The question of how fast is, in a easier to answer. To achieve greatest performance, the memory must be able to up with the processor. That is, as the processor is executing instructions, we not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be able in relationship to other components.
As might be expected, there is a trade-off among the three key character of memory: namely, capacity, access time, and cost. A variety of technologic used to implement
memory systems, and across this spectrum of technology following relationships hold:
Faster access time, greater cost per bit
Greater capacity, smaller cost per bit
Greater capacity, slower access time
The dilemma facing the designer is clear. The designer would like to use memory
technologies that provide for large-capacity memory, both because the car
is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a memory hierarchy. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:
Decreasing cost per bit i > Increasing capacity
7a Increasing access time
Decreasing frequency of access of the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.





Example 4.1 Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of 0.01 μs; level 2 contains 100,000 words and has an access time of 0.1 μs. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Figure 4.2 shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[1] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in the cache. Then the average time to access a word can be expressed as

$$(0.95)(0.01\ \mu\text{s}) + (0.05)(0.01\ \mu\text{s} + 0.1\ \mu\text{s}) = 0.0095 + 0.0055 = 0.015\ \mu\text{s}$$

The average access time is much closer to 0.01 μs than to 0.1 μs, as desired.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference** [DENN68]. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2 memory contain all program instructions and data. The current clusters can be temporarily placed in level l. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 4.1. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a -processor will contain a few dozen such registers, although some machines contain hundreds of registers. Skipping down two levels, main memory is the principal internal memory system of the computer. Each location in main memory has a unique _address. Main memory is usually extended with a higher-speed, smaller cache. The -ache is not usually visible to the programmer or, indeed, to the processor. It is a de¯.¯ice for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ miconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most com on are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as secondary memory auxiliary memory. These are used to store program and data files and are usually invisible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory own as virtual memory, which is discussed in Chapter 8.

Other forms of memory may be included in the hierarchy. For example, large 1 mainframes include a form of internal memory known as expanded storage is uses a semiconductor technology that is slower and less expensive than that of in memory. Strictly speaking, this memory does not fit into the hierarchy but is a branch: Data can be moved between main memory and expanded storage but between expanded storage and external memory. Other forms of secondary memory include optical and magneto-optical disks. Finally, additional levels can be positively added to the hierarchy in software. A portion of main memory can be used as a buffer to hold data temporarily that is to be read out to disk. Such a technique, sometimes referred to as a disk cache, [2] improves performance in two ways

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimize processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

ERROR CORRECTION

A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A hard failure is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A soft error is a random, nondestructive event that alters the contents of one or more memory cells without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials. Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 5.7 illustrates in general terms how the process is carried out. When data are to be read into memory, a calculation, depicted as a function f, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an _VI-bit word of data is to be stored and the code is of length *K* bits, then the actual size of the stored word is M + *K* bits.

When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of *K* code bits is generated from the M data bits and compared with the fetched code bits. The comparison yields one of three results:

No errors are detected. The fetched data bits are sent out.

An error is detected, and it is possible to correct the error. The data bits plus error correction bits are fed into a corrector, which produces a corrected set of M bits to be sent out.

An error is detected, but it is not possible to correct it. This condition is reported.

Codes that operate in this fashion are referred to as *error-correcting codes. A* code is characterized by the numbs of bit errors in a word that it can correct and detect.

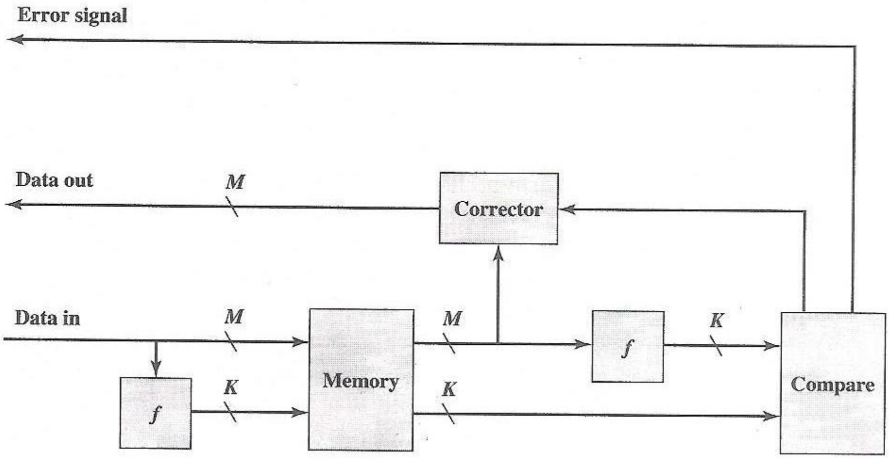
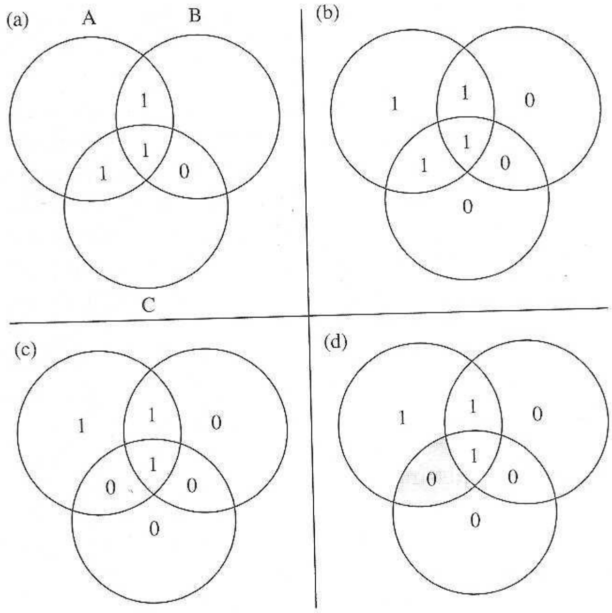


Figure 5.7 Error-Correcting Code Function



The simplest of the error-correcting codes is the *Hamming code* devised b: Richard Hamming at Bell Laboratories. Figure 5.8 uses Venn diagrams to illustrate the use of this code on 4-bit words (M = 4). With three intersecting circles, there aseven compartments. We assign the 4 data bits to the inner compartments (Figure

5.8a). The remaining compartments are filled with what are called *parity b-1*Each parity bit is chosen so. that the total number of is in its circle is even (Figure 5.8b). Thus, because circ A includes three data 1s, the parity bit in that circle set to 1. Now, if an error changes one of the data bits (Figure 5.8c), it is easily four

By checking the parity bits, discrepancies are found in circle A and circle C but in circle B. Only one of the seven compartments is in A and C but not B. The er can therefore be corrected by changing that bit.

To clarify the concepts involved, we will develop a code that can detect G^{-} - correct single-bit errors in 8-bit words.

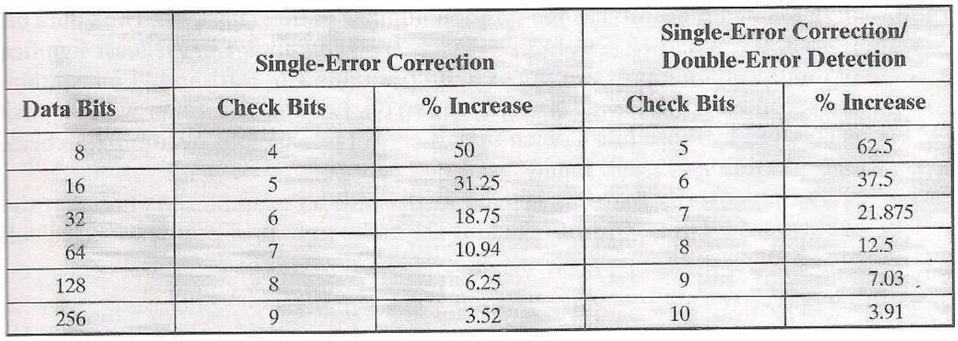
To start, let us determine how long the code must be. Referring to Figure - the comparison logic receives as input two K-bit values. A bit-by-bit comparison done by taking the exclusive-OR of the two inputs. The result is called the *synod_ word. Thus,* each bit of the syndrome is 0 or 1 according to if there is or is n,_ - match in that bit position for the two inputs.

The syndrome word is therefore *K* bits wide and has a range between C. The value 0 indicates that no error was detected, leaving 2^K - 1 value - indicate, if there is an error, which bit was in error. Now, because an error c,

occur on any of the M data bits or *K* check bits, we must have

2K-1>>_M+K

Table 5.2 Increase in Word Length with Error Correction



	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	Check Bits	% Increase	Check Bits	% Increase
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

1.0 Conclusion

As one goes down the memory hierarchy one finds decreasing cost bit, increasing capacity, and slower access time. This unit focuses on internal memory elements.

2.0 Summary

Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance and cost of any feature of computer system

The error correction technique involves adding redundant bits that are a function of the data bit to form an error correction code. If a bits error occurs, the code will detect and usually correct the error.

3.0 Tutor Marked Assignment

1. What are the differences among direct mapping, associative mapping and set associative mapping?
2. What is a parity bit?
3. How is the syndrome for the hamming code interpreted?

4.0 References/Further reading

1. Adamck, J. Foundation of coding New York Wiley 1991
2. Smith,a CACHE MEMORIES ACM computing surveys September 1992

UNIT 4: CACHE MEMORY

1.0 Introduction

In general it is likely that most future access to main memory by the processor will be to locations recently accessed. So the cache memory automatically retains a copy of some of the recently used words form the dynamic random- access memory (DRAM)

2.0 Objectives

At the end of this unit, you should be able to

- Explain the principles and elements of cache design\understood Pentium 4 cache organization
- Discuss ARM cache organization

3.1 Cache memory principles

3.2 Elements of cache design

3.3 Pentium 4 cache organization

3.4 ARM cache organization

4.0 Conclusion

If the cache is designed properly then most of the time the processor will request memory words that are already in the cache.

5.0 SUMMARY

Cache memories are intended to give memory speeds and large memory size. Besides, cache are often used in high performance computers that deals with super computers and supercomputers software for scientific application. Moreover the longer the cache the larger the number of gates involved in addressing the cache.

6.0 Tutor marked assignment

For a direct mapped cache a main memory address is viewed as consisting of two fields list and define the two fields.

7.0 Reference and further reading

Agarwal, A. Analysis of cache performance for operating systems and multiprogramming. Boston: Kluwer academic publishers 1989.

MODULE 6 LOGIC

UNIT 1: BOOLEAN ALGEBRA

UNIT 2: LOGIC OPERATIONS

UNIT 3: COMBINATIONAL CIRCUITS

UNIT 1: BOOLEAN ALGEBRA

Introduction

1.0 Objectives
2.0 Main content
    2.1 Digital circuitry
    2.2 Boolean operators
    2.3 Basic identities of Boolean Algebra
3.0 Conclusion
4.0 Summary
5.0 Tutor marked assignment
6.0 References/ Further reading

Construct a truth table for the following Boolean expression:

a. ABC+ ABC
b. ABC + ABC + ABC
c. A (BC + BC)
d. (A +B) (A+C) (A+B)

7.0 References/ further reading

The logic of sets Gregg.Jones and zeros: Understanding Boolean Algebra, Digital

1.0 INTRODUCTION

The operation of the digital computer is based on the storage and processing of binary data. In this unit we suggest how storage elements and circuits can be implemented in digital logic specifically with combinational and sequential circuits.

2.0 OBJECTIVES

At the end of this unit you should be able to

- Understand Boolean operators
- Explain the Boolean operators
- Discuss the identities of Boolean Algebra

3.1 The digital circuitry
3.2 Explain the Boolean operators
3.3 The basic

3.1 DIGITAL CIRCUITRY

The digital circuitry in digital computers and other digital systems is designed, and its behavior is analyzed, with the use of a mathematical discipline known as Boolean algebra. The name is in honor of an English mathematician George Boole, who proposed the basic principles of this algebra in 1854 in his treatise,

An Investigation of the Laws of Thought on Which to Found the Mathematical Theories of Logic and Probabilities. In 1938, Claude Shannon, a research assistant in the Electrical Engineering Department at M.I.T, suggested that Boolean algebra could be used to solve problems in relay-switching circuit design [SFIAN38].[f] Shannon's techniques were subsequently used in the analysis and design of electronic digital circuits. Boolean algebra turns out to be a convenient tool in two areas:

Analysis: It is an economical way of describing the function of digital circuitry.

Design: Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

As with any algebra, Boolean algebra makes use of variables and operations. In this case, the variables and operations are logical variables and operations. Thus, a variable may take on the value 1 (TRUE) or 0 (FALSE). The basic logical operations are AND, OR, and NOT, which are symbolically represented by dot, plus sign, and over bar:[2]

A AND B=A-B
A OR B=A+B
NOT A = A

The operation AND yields true (binary value 1) if and only if both of its operands are true. The operation OR yields true if either or both of its operands are true. The unary operation NOT inverts the value of its operand. For example, consider the equation

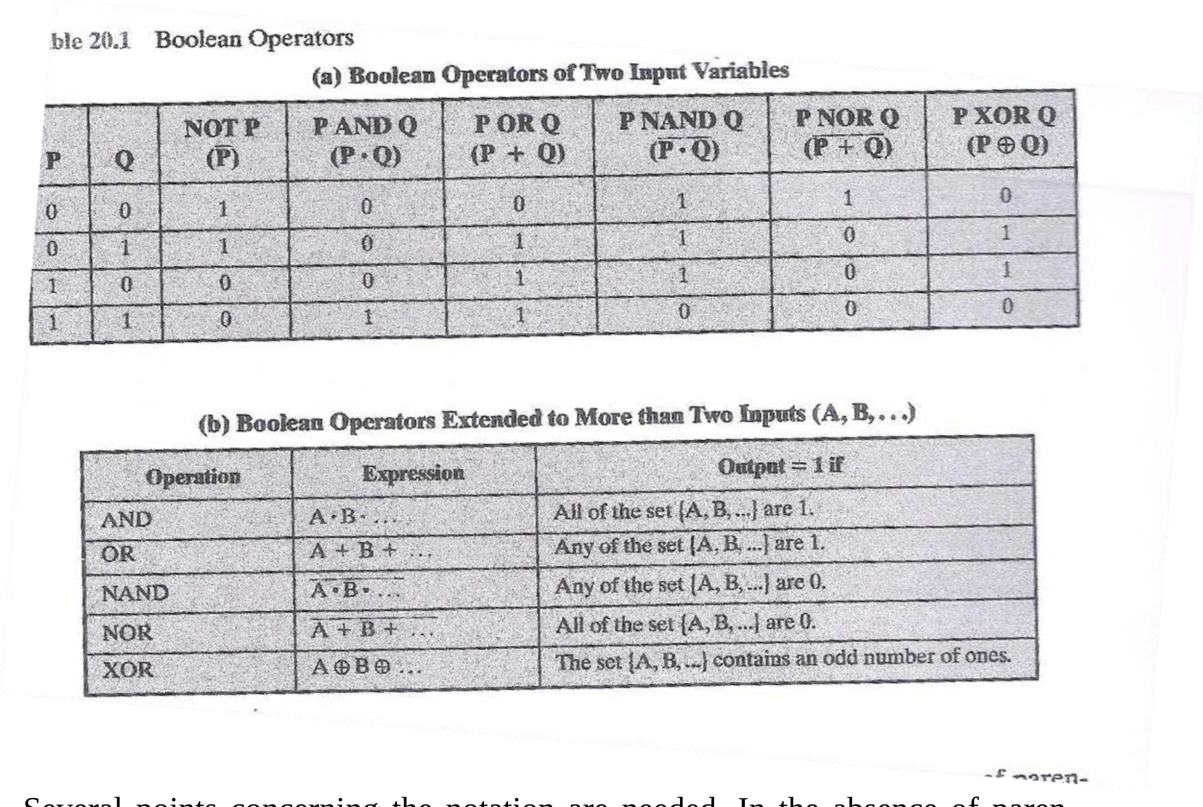
$$D=A+(B'C)$$

D is equal to 1 if A is 1 or if both B = 0 and C = 1. Otherwise D is equal to 0.

'The paper is available at this book's web site.

[2] Logical NOT is often indicated by an apostrophe: NOT A = A'.

3.2 BOOLEAN OPERATORS



ble 20.1 Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P $(\overline{P})$	P AND Q $(P \cdot Q)$	P OR Q $(P + Q)$	P NAND Q $(\overline{P \cdot Q})$	P NOR Q $(\overline{P + Q})$	P XOR Q $(P \oplus Q)$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, ...} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, ...} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, ...} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, ...} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, ...} contains an odd number of ones.

Several points concerning the notation are needed. In the absence of parentheses, the AND operation takes precedence over the OR operation. Also, when no ambiguity will occur, the AND operation is represented by simple concatenation instead of the dot operator. Thus,

A+B-C=A+(B-C)=A+BC

all mean: Take the AND of B and C; then take the OR of the result and A.

Table 20.1a defines the basic logical operations in a form known as a truth table, which lists the value of an operation for every possible combination of values of operands. The table also lists three other useful operators: XOR, NAND, and NOR. The exclusive-or (XOR) of two logical operands is 1 if and only if exactly one of the operands has the value 1. The NAND function is the complement (NOT) of the AND function, and the NOR is the complement of OR:

A NAND B = NOT (A AND B)$^{-}$- AB

A NOR B = NOT (A OR B) = A + B

As we shall see, these three new operations can be useful in implementing certain digital circuits.

3.3 THE BASIC IDENTITIES OF BOOLEAN ALGEBRA

Table 24.2 Basic Identities of Boolean Algebra

Basic Postulates		
A - B = B - A	A -t- B = B + A	Commutative
A - (B + C) _ (A-13) + (A	-A + (B - C) - (A + B)-(A + C)	Distributive Laws
1- A = A	0 + A = A	Identity Elements
A- A = 0	A-1 A = 1	Inverse Elements
Other Identities		
o-A=0	1 +A =I	
A-A=A	Aj-A-A	
A - (B -C') -- (A - B) - C	A + (B + C) -- (A + B) + C	Associative Laws
X^{-}-$^{-}$B - A + B	A -= B = A -13	DeMorgan's Theorem

The logical operations, with the exception of NOT, can be generalized to more than two variables, as shown in Table 20.1b.

Table 20.2 summarizes key identities of Boolean algebra. The equations have been arranged in two columns to show the complementary, or dual, nature of the AND and OR operations. There are two classes of identities: basic rules (orpostulates), which are stated without proof and other identities that can be derived from the basic postulates. The postulates define the way in which Boolean expressions are interpreted. One of the two distributive laws is worth noting because it differs from what we would find in ordinary algebra:

$$A+(B^{\circ}C)-(A+B)-(A\underline{+}C)$$

A HAND B = A OR R

The reader is invited to verify the expressions in Table 20.2 by substituting actual values (1s and Os) for the variables A, B, and C.

UNIT 2: LOGIC OPERATIONS

1.0 INTRODUCTION

The fundamental building block of all digital logic circuits is the gate. Logical functions are implemented by the interconnection of gates.

2.0 OBJECTIVES

At the end of this unit, you should be able to

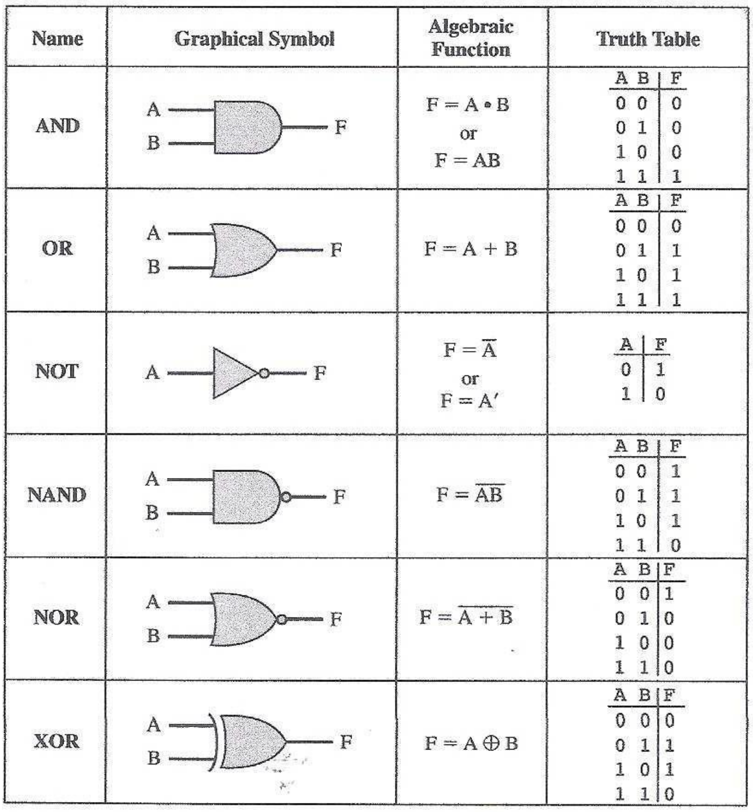
- Explain the basic logic gates
- Design by the line diagram for each gates/logical circuits

Construct truth table of logical circuits

3.1 GATES

A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals. The basic gates used in digital logic are AND, OR, NOT, NAND, NOR, and XOR. Figure 20.1 depicts these six gates. Each gate is defined in three ways: graphic symbol, algebraic notation, and truth table. The symbology used here and throughout the appendix is the IEEE standard, IEEE Std 91. Note that the inversion (NOT) operation is indicated by a circle.

Each gate shown in Figure 20.1 has one or two inputs and one output. However, as indicated in Table 20.1b, all of the gates except NOT can have more than two inputs. Thus, (X + Y + Z) can be implemented with a single OR gate with three inputs. When one or more of the values at the input are changed, the correct output signal appears almost instantaneously, delayed only by the propagation time of



Name	Graphical Symbol	Algebraic Function	Truth Table
AND	A B F	$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	A B F	$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT	A F	$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND	A B F	$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR	A B F	$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR	A B F	$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

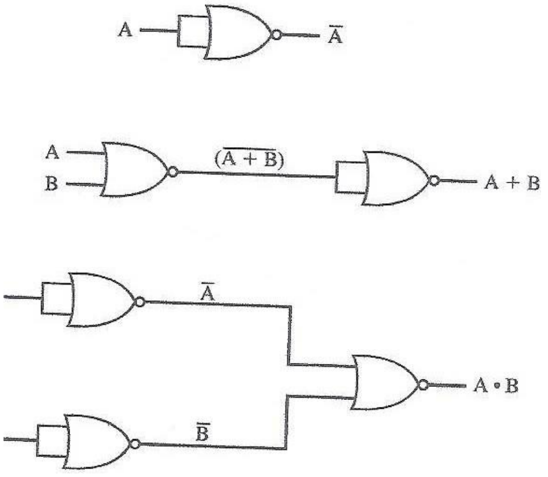
signals through the gate (known as the gate delay). The significance of this delay is discussed in Section 20.3. In some cases, a gate is implemented with two outputs, one output being the negation of the other output.

Here we introduce a common term: we say that to assert a signal is to cause signal line to make a transition from its logically false (0) state to its logically true (1) state. The true (1) state is either a high or low voltage state, depending on the type of electronic circuitry.

3.2 BASIC LOGIC GATES

Typically, not all gate types are used in implementation. Design and fabrication are simpler if only one or two types of gates are used. Thus, it is important to identify *functionally complete* sets of gates. This means that any Boolean function can be implemented using only the gates in the set. The following are functionally complete sets:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR



It should be clear that AND, OR, and NOT gates constitute a functionally complete set, because they represent the three operations of Boolean algebra. For the AND and NOT gates to form a functionally complete set, there must be a way to synthesize the OR operation from the AND and NOT operations. This can be done by applying DeMorgan's theorem:

$A+B=A-B$

A OR B = NOT (.(NOT A) AND (NOT B))

Similarly, the OR and NOT operations are functionally complete because they can be used to synthesize the AND operation.

Figure 211.2 shows how the AND, OR, and NOT functions can be implemented solely with HAND gates, and Figure 20.3 shows the same thing for NOR gates. For this reason, digital circuits can be, and frequently are, implemented solely with NAND gates or solely with NOR gates.

With gates, we have reached the most primitive circuit level of computer hardware. An examination of the transistor combinations used to construct gates departs from that realm and enters the realm of electrical engineering. For our purposes, however, we are content to describe how gates can be used as building blocks to implement the essential logical circuits of a digital computer

4.0 CONCLUSION

Typically not all gates types are used. The design and fabrication of circuits becomes simpler and earlier of one or two gates are used.

5.0 SUMMARY

In summary, we have size basic logic gates which are defined in three ways, namely graphic symbol, algebraic notation and truth table.

6.0 TUTOR MARKED ASSIGNMENT

1. Given a NOR gate and NOT gates, draw a logic diagram that will perform the tree input and function.
2. Construct the operations XOB from the basic Boolean operations and OR na Dnot

7.0 REFERENCES/ FURTHER READING

Farahat, H. Digital design and computer organization Boca Ratan: CRC press, 2004

UNIT 3: COMBINATIONAL CIRCUITS

1.0 Introduction
2.0 Objectives
3.0 Main content
3.1 Implementation of Boolean functions

1.0 Introduction

A combinational circuit is an interconnected set of gates whose output at any time is a function only of the output at that time. As with a single gate, the appearance of the input is followed almost immediately by the appearance of the output with only gate delays.

2.0 Objectives

At the end of this unit, you should be able to

- Understand the truth table and graphical symbols of a combinational circuits with its Boolean equations inclusive.
- Explain and discuss algebraic simplification, karnaugh maps and quine mckluskey tables

3.1 IMPLEMENTATION OF BOOLEAN FUNCTIONS

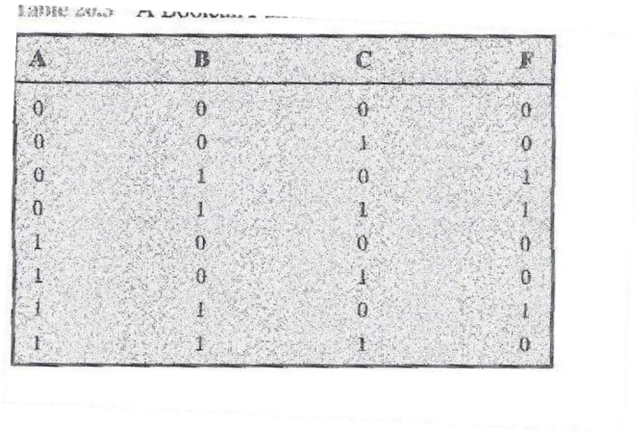
In general terms, a combinational circuit consists of n binary inputs and m binary outputs. As with a gate, a combinational circuit can be defined in three ways:

- **Truth table:** For each of the 2" possible combinations of input signals, the binary value of each of the m output signals is listed.
- **Graphical symbols:** The interconnected layout of gates is depicted.
- **Boolean equations:** Each output signal is expressed as a Boolean function of its input signals.

Any Boolean function can be' implemented in electronic form as a network of gates. For any given function, there are a number of alternative realizations. Consider the Boolean function represented by the truth table in Table 20.3. We can *express* this function by simply itemizing the combinations of values of A, B, and C that cause F to be 1:

$$F = ABC + ABC + ABC \qquad (20.1)$$

There are three combinations of input values that cause F to be 1, and if any one of these combinations occurs, the result is 1. This form of expression, for self-evident



A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Sum-of-Products Implementation of Table 20.3

reasons, is known as the sure of products (SOP) form: Figure 20.4 shows a straightforward implementation with AND, OR, and NOT gates.

Another form can also be derived from the truth table. The SOP form expresses that the output is 1 if any of the input combinations that produce 1 is true. We can also say that the output is 1 if none of the input combinations that produce 0 is true. Thus,

F = (A g C) _ (A g C) _ (ABC) - (ABC) - (ABC)

This can be rewritten using a generalization of DeMorgan's theorem:
(X•Y.Z)=X+Y+Z

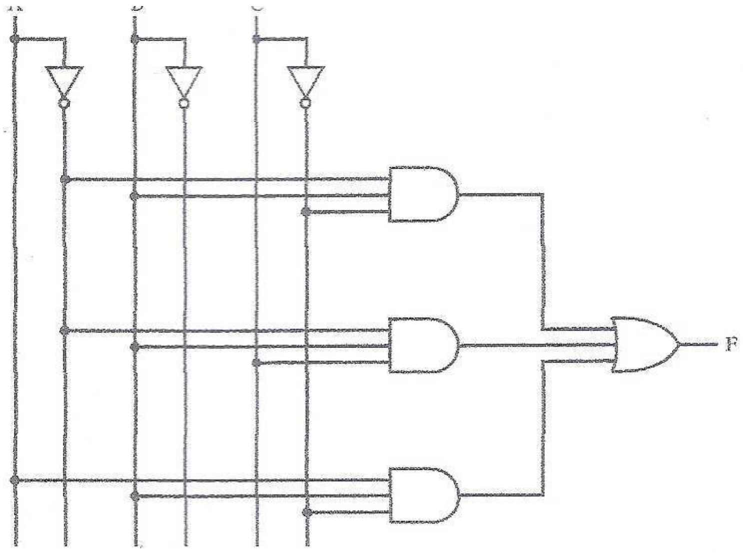
Thus,

F=(A+B+C)-(A+B+c)-(A+B+c)(A+B+C)-(A+B+C) (lox) =(A+B+C)-(A+B+c)-(A+B+C)-(A+B+c)-(A+R+c)

This is in the product of sums (POS) form, which is illustrated in Figure 20.5. For clarity, NOT gates are not shown. Rather, it is assumed that each input signal and its complement are available. This simplifies the logic diagram and makes the inputs to the gates more readily apparent.

Thus, a Boolean function can be realized in either SOP or POS form. At this point, it would seem that the choice would depend on whether the truth table contains more 1s or Os for the output function: The SOP has one term for each 1, and the POS has one term for each 0. However, there are other considerations:

It is often possible to derive a simpler Boolean expression from the truth table than either SOP or POS.



Implementation of Table 20.3

It may be preferable to-implement the function with a single gate type (NAND or NOR).

The significance of the first point is that, with a simpler Boolean expression, fewer gates will be needed to implement the function. Three methods that can be used to achieve simplification are Algebraic simplification Karnaugh maps Quine-VIcKluskey tables.

3.2 SIMPLIFICATION OF BOOLEAN FUNCTIONS

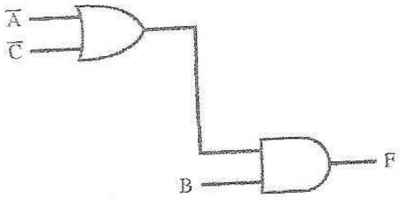
Algebraic simplification involves the application of the identities of Table 20.2 to reduce the Boolean expression to one with fewer elements. For example, consider again Equation (20.1). Some thought should convince the reader that an equivalent expression is

$F = AB + BC$ (20.3)

Or, even simpler,

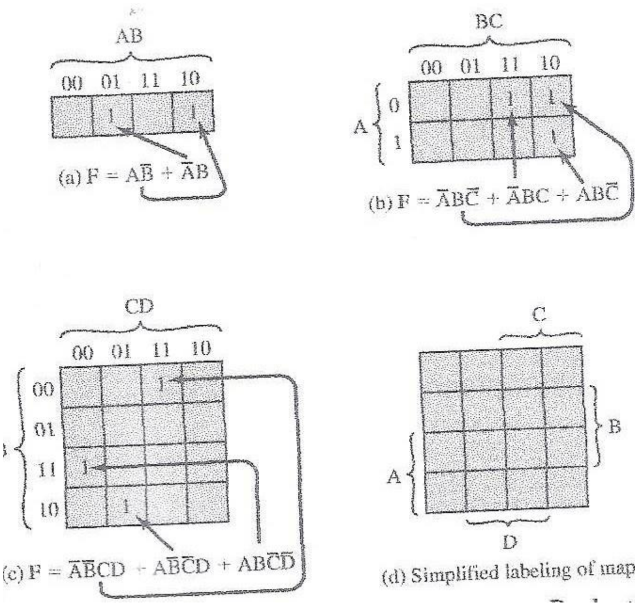
F=B(A+~)

This expression can be implemented as shown in Figure 20.6. The simplification of Equation (20.1) was done essentially by observation. For more complex expressions, some more systematic approach is needed.



For purposes of simplification, the Karnaugh map is a convenient way of representing a Boolean function of a small number (up to four) of variables. The map is an array of 2" squares, representing all possible combinations of values of n binary variables. Figure 20.7a shows the :nap of four squares for a function of two variables. It is essential for later purposes to list the combinations in the order 00, 01, Because the squares corresponding to the combinations are to be used for recording information, the combinations are customarily written above the squares. In the case of three variables, the representation is an arrangement of eight squares (Figure 20.7b), with the values for one of the variables to the left and for the other two variables above the squares. For four variables, 16 squares are needed, with the arrangement indicated in Figure 20.7c.

The map can be used to represent any Boolean function in the following way. Each square corresponds to a unique product in the sum-of-products form, with a 1 value corresponding to the variable and a 0 value corresponding to the NOT of that



(a) F = $A\overline{B} + \overline{A}B$

(b) F = $\overline{A}B\overline{C} + \overline{A}BC + AB\overline{C}$

(c) F = $\overline{A}\overline{B}CD + A\overline{B}\overline{C}D + AB\overline{C}\overline{D}$

(d) Simplified labeling of map

variable. Thus, the product AB corresponds to the fourth square in Figure 20.7a. For each such product in the function, 1 is placed in the corresponding square. Thus, for the two-variable example, the map corresponds to AB + WB. Given the truth table of a Boolean function, it is an easy matter to construct the map: for each combination of values of variables that produce a result of 1 in the truth table, fill in the corresponding square of the map with 1. Figure 20.7b shows the result for the truth table of Table 20.3. To convert from a Boolean expression to a map, it is first necessary to put the expression into what is referred to as *canonical* form: each term in the expression must contain each variable. So, for example, if we have Equation (20.3), we must first expand it into the full form of Equation (20.1) and then convert this to a map.

The labeling used in Figure 20.7d emphasizes the relationship between variables and the rows and columns of the map. Here the two rows embraced by the symbol A are those in which the variable A has the value 1; the rows not embraced by the symbol A are those in which A is 0; similarly for B, C, and D.

Once the map of a function is created, we can often write a simple algebraic expression for it by noting the arrangement of the 1s on the map. The principle is as follows. Any two squares that are adjacent differ in only one of the variables. If two adjacent squares both have an entry of one, then the corresponding product terms differ in only one variable. In such a case, the two terms can be merged by eliminating that variable. For_ example, in Figure 20.8a, the two adjacent squares correspond to the two terms ABCD and ABCD. Thus, the function expressed is

ABCD + ABCD = ABD

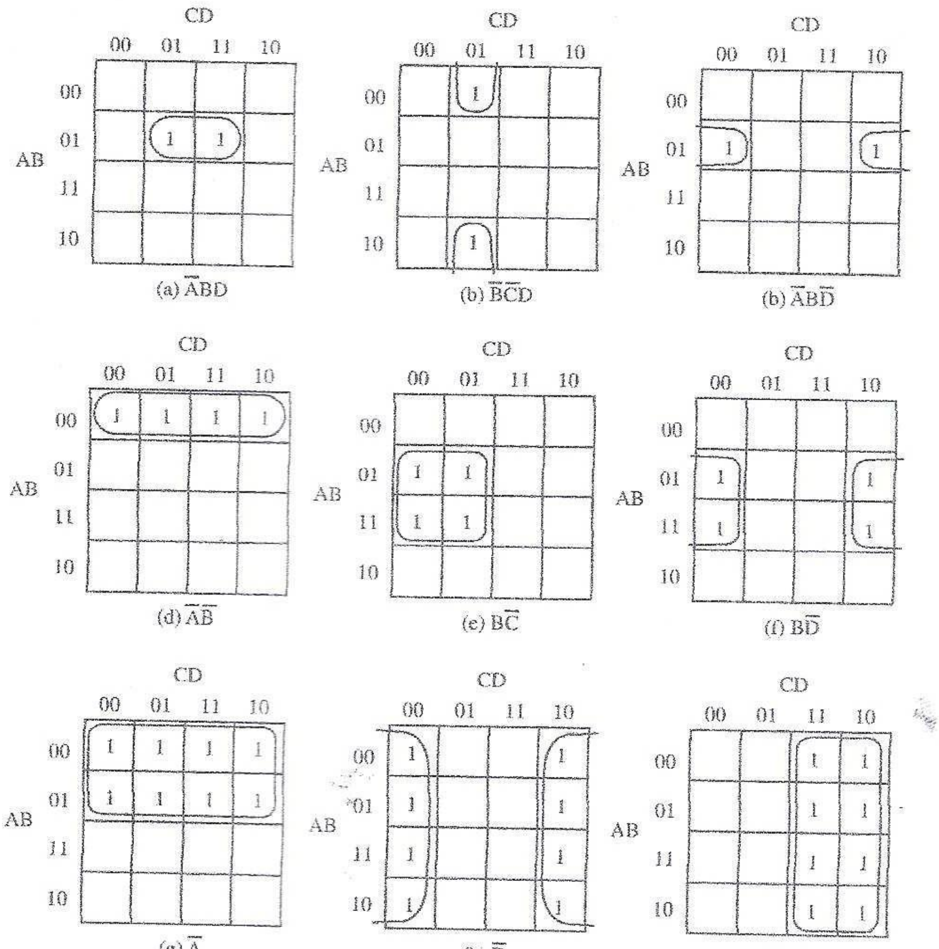
This process can be extended in several ways. First, the concept of adjacency can be extended to include wrapping around the edge of the map. Thus, the top square of a column is adjacent to the bottom square, and the leftmost square of a row is adjacent to the rightmost square. These conditions are illustrated in Figures 20.8b and c. Second, we can group not just 2 squares but 2'Z adjacent squares (that is, 2, 4, 8, etc.). The next three examples in Figure 20.8 show groupings of 4 squares. Note that in this case, two of the variables can be eliminated. The last three examples show groupings of 8 squares, which allow three variables to be eliminated.

We can summarize the rules for simplification as follows:

Among the marked squares (squares with a 1), find those that belong to a unique largest block of 1, 2, 4, or 8 and circle those blocks.

Select additional blocks of marked squares that are as large as possible and as few in number as possible, but include every marked square at least once. The results may not be unique in some cases. For example, if a marked square combines with exactly two other squares, and there is no fourth marked square to complete a larger group, then there is a choice to be made as two which of the two groupings to choose. When you are circling groups, you are allowed to use the same 1 value more than once.

Continue to draw loops around single marked squares, or pairs of adjacent marked squares, or groups of four, eight, and so on in such a way that every marked square belongs to at least one loop; then use as few of *these* blocks as possible to include all marked squares.



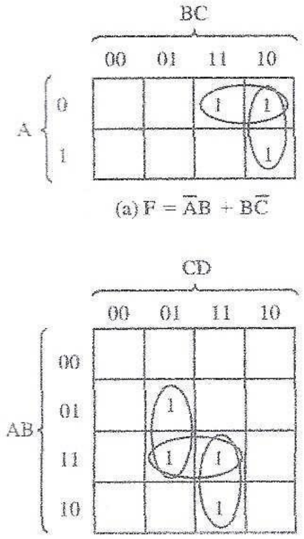
(a) $\overline{A}BD$ (b) $\overline{B}\overline{C}D$ (b) $\overline{A}B\overline{D}$

(d) $\overline{A}\overline{B}$ (e) $B\overline{C}$ (f) $B\overline{D}$

(g) $\overline{A}$ (h) $\overline{D}$ (i) C

Figure 20.9x, based on Table 20.3, illustrates the simplification process. If any isolated 1s remain after the groupings, then each of these is circled as a group of 1s. Finally, before going from the map to a simplified Boolean expression, any group of 1s that is completely overlapped by other groups can be eliminated. This is shown in Figure 20.9b. In this case, the horizontal group is redundant and may be ignored in creating the Boolean expression.

One additional feature of Karnaugh maps needs to be mentioned. In some cases, certain combinations of values of variables never occur, and therefore the corresponding output never occurs. These are referred to as "don't care" conditions. For each such condition, the letter "d" is entered into the corresponding square of the map. In doing the grouping and simplification each “d” can be treated as 1 or 0 whichever leads to the simplest expression.



(a) F = $\overline{A}B + B\overline{C}$

(b) F = $B\overline{C}D + ACD$

Figure 20.9 Overlapping Groups

An example, presented in [HAYE98], illustrates the points we have been discussing. We would like to develop the Boolean expressions for a circuit that adds 1 to a packed decimal digit. Recall from Section 9.2 that with packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way. Thus, 0 = 0000, 1 = 0001, . . ., 8 = 1000, and 9 = 1001. The remaining 4-bit values, from 1010 to 1111, are not used. This code is also referred to as Binary Coded Decimal (BCD).

Table 20.4 shows the truth table for producing a 4-bit result that is one more than a 4-bit BCD input. The addition is modulo 10. Thus, 9 + 1 = 0. Also, note that six of the input codes produce "don't care" results, because those are not valid BCD inputs. Figure 20.10 shows the resulting Karnaugh maps for each of the output variables. The d squares are used to achieve the best possible groupings.

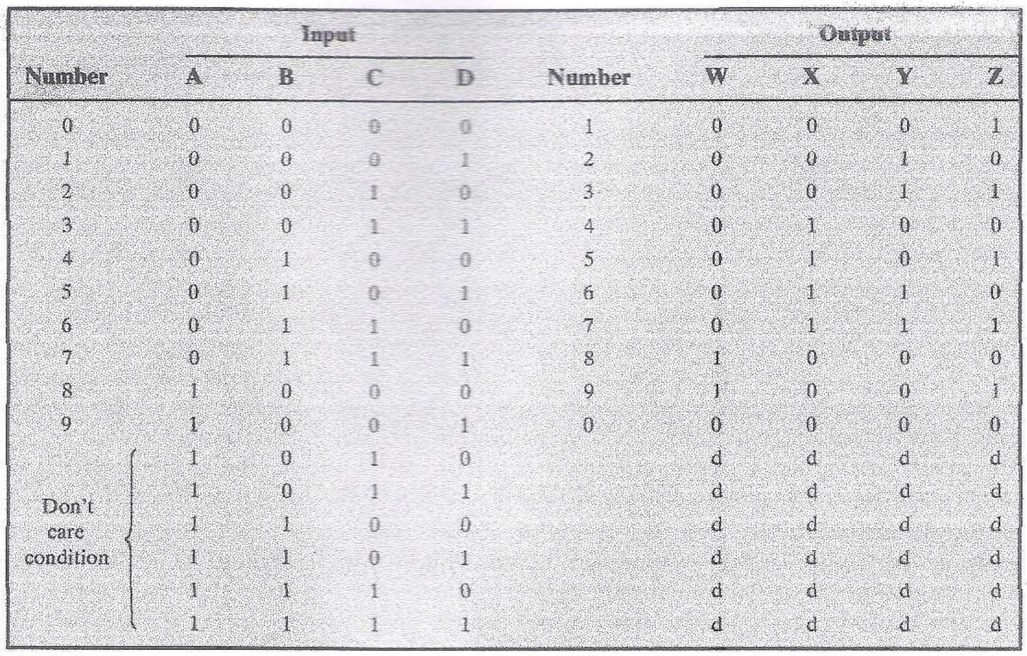
For more than four variables, the Karnaugh map method becomes increasingly cumbersome. With five variables, two 16 x 16 maps are needed, with one map considered to be on top of the other in three dimensions to achieve adjacency. Six variables require the use of four 16 x 16 tables in four dimensions! An alternative approach is a tabular technique, referred to as the Quine-McKluskey method. The method is suitable for programming on a computer to give an autoirmafic fool for producing minimized Boolem expregSiong.

The method is best explained by means of an example. Consider the following expression:

ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD

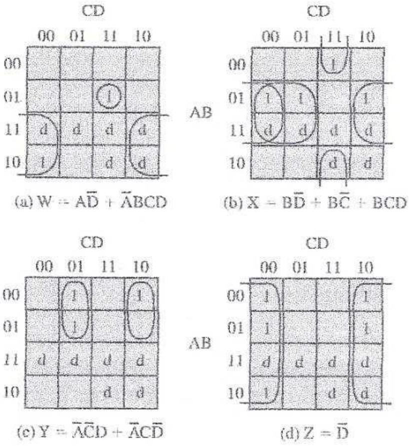
Let us assume that this expression was derived from a truth table. We would like to produce a minimal expression suitable for implementation with gates.

Table 20.4 Truth Table for the One-Digit Packed Decimal Incrementer



	Input					Output			
Number	A	B	C	D	Number	W	X	Y	Z
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	2	0	0	1	0
2	0	0	1	0	3	0	0	1	1
3	0	0	1	1	4	0	1	0	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	6	0	1	1	0
6	0	1	1	0	7	0	1	1	1
7	0	1	1	1	8	1	0	0	0
8	1	0	0	0	9	1	0	0	1
9	1	0	0	1	0	0	0	0	0
Don't care condition	1	0	1	0		d	d	d	d
	1	0	1	1		d	d	d	d
	1	1	0	0		d	d	d	d
	1	1	0	1		d	d	d	d
	1	1	1	0		d	d	d	d
	1	1	1	1		d	d	d	d

The first step is to con_ trLictt a table in which each row corresponds to one of the product terms of the expression. The terms are grouped according to the number of complemented variables. I =:at is; eve start with the term with no complements, if it exists, then all terms with one complement, and so on. Table 20.5 shows the list for our example ***expression,* with horizontal fines** used to indicate the grouping. For clarity,



AB \ CD	00	01	11	10
00				
01			1	
11	d	d	d	d
10	1		d	d

(a) $W = A\overline{D} + \overline{A}BCD$

AB \ CD	00	01	11	10
00			1	
01	1	1		1
11	d	d	d	d
10			d	d

(b) $X = B\overline{D} + B\overline{C} + BCD$

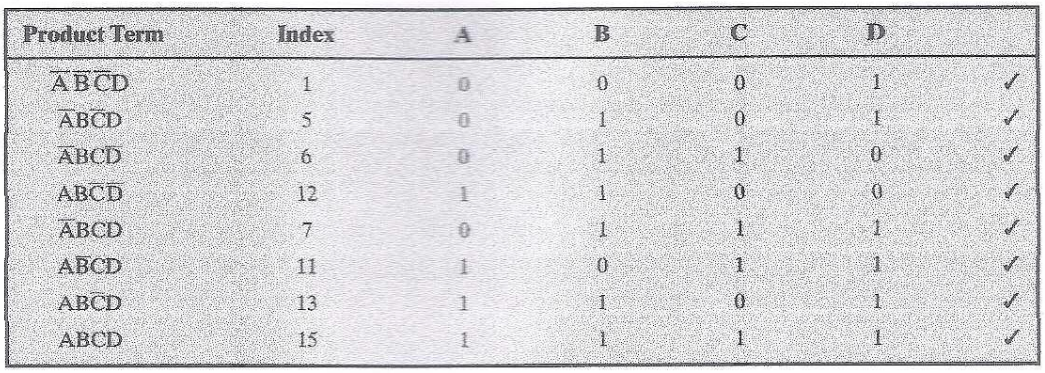
AB \ CD	00	01	11	10
00		1		1
01		1		
11	d	d	d	d
10			d	d

(c) $Y = \overline{A}\overline{C}D + \overline{A}C\overline{D}$

AB \ CD	00	01	11	10
00	1			1
01	1			1
11	d	d	d	d
10	1		d	d

(d) $Z = \overline{D}$

Table 20.5 First Stage of Quine-McKluskey Method
(for $F = ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$)



Product Term	Index	A	B	C	D	
$\overline{A}\,\overline{B}\,\overline{C}D$	1	0	0	0	1	✓
$\overline{A}B\overline{C}D$	5	0	1	0	1	✓
$\overline{A}BC\overline{D}$	6	0	1	1	0	✓
$AB\overline{C}\,\overline{D}$	12	1	1	0	0	✓
$\overline{A}BCD$	7	0	1	1	1	✓
$A\overline{B}CD$	11	1	0	1	1	✓
$AB\overline{C}D$	13	1	1	0	1	✓
$ABCD$	15	1	1	1	1	✓

each term is represented by a 1 for each uncomplemented variable and a 0 for each complemented variable. Thus, we group terms according to the number of is they contain. The index column is simply the decimal equivalent and is useful in what follows.

The next step is to find all pairs of terms that differ in only one variable, that is, all pairs of terms that are the same except that one variable is 0 in one of the terms and 1 in the other. Because of the way in which we have grouped the terms, we can do this by starting with the first group and comparing each term of the first group with every term of the second group. Then compare each term of the second group with all of the terms of the third group, and so on. Whenever a match is found, place a check next to each term, combine the pair by eliminating the variable that differs in the two terms, and add that to. a new list. Thus, for example, the terms ABCD and ABCD are combined to prodai&e ABC. This process continues until the entire original table has been examined. The result is a new table with the following entries:

A CD ABC ABD ~' $BCD_{\%}$ ACD

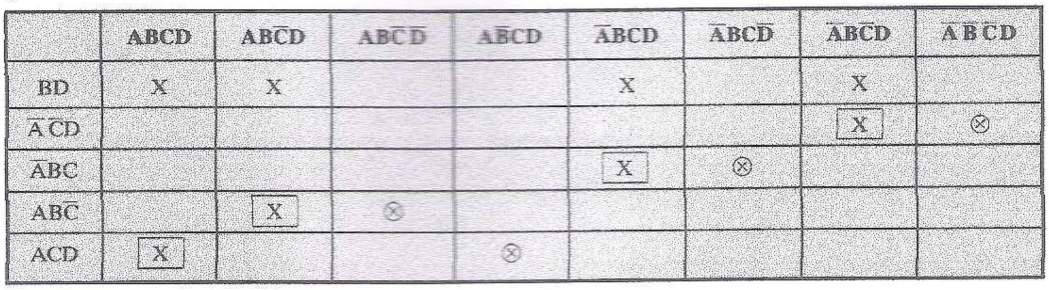
ABC BCD ABD_{V}

The new table is organized into groups, as indicated, in the same fashion as the first table. The second table is then processed in the same manner as the first. That is, terms that differ in only one variable are checked and a new term produced for a third table. In this example, the third table that is produced contains only one term: BD.

In general, the process would proceed through successive tables until a table with no matches was produced. In this case, this has involved three tables.

Once the process just described is completed, we have eliminated many of the possible terms of the expression. Those terms that have not been eliminated are used to construct a matrix, as illustrated in Table 20.6. Each row of the matrix corresponds to one of the terms that have not been eliminated (has no check) in any of the tables used so far. Each column corresponds to one of the terms in the original expression. An X is placed at each intersection of a row and a column such that the row element is "compatible" with the column element. That is, the variables present in the row element have the same value as the variables present in the column element. Next, circle each X that is alone in a column. Then place a square around each X in any row in which there is a circled X. If every column now has either a squared or a circled X, then we are done, and those row elements whose Xs have been marked constitute the minimal expression. Thus, in our example, the final expression is

Table 20.6 Last Stage of Quine-McKluskey Method
(for $F = ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$)



	$ABCD$	$AB\overline{C}D$	$AB\overline{C}\,\overline{D}$	$A\overline{B}CD$	$\overline{A}BCD$	$\overline{A}BC\overline{D}$	$\overline{A}B\overline{C}D$	$\overline{A}\,\overline{B}\,\overline{C}D$
BD	X	X			X		X	
$\overline{A}\,\overline{C}D$							$\boxed{\text{X}}$	⊗
$\overline{A}BC$					$\boxed{\text{X}}$	⊗		
$AB\overline{C}$		$\boxed{\text{X}}$	⊗					
ACD	$\boxed{\text{X}}$			⊗				

A B C ! + A C D + I B C + X U D

In cases in which some columns have neither a circle nor a square, additional processing is required. Essentially, we keep adding row elements until all columns are covered.

Let us summarize the twine-McKluskey method to try to justify intuitively why it works. The first phase of the operation is reasonably straightforward, *The* process eliminates unneeded variables in product terms. Thus, the expression ABC + ABC is equivalent to AB; because

ABC + ABC = AB(C + C) = A

After the elimination of variables, we are left with an expression that is clearly equivalent to the original expression. However, there may be redundant terms in this expression, just as we found redundant groupings in Karnaugh maps. The matrix layout assures that each term in the original expression is covered and does so in a way that minimizes the number of terms in the final expression.

Another consideration in the implementation of Boolean functions concerns the types of gates used. It is sometimes desirable to implement a Boolean function solely with TeTAND gates or solely with NOR gates. Although this may not be the minimum-gate implementation, it has the advantage of regularity, which can simplify the manufacturing process. Consider again Equation (2(1.3):

Because the complement of the complement of a value is just the original value;

F=B(A+C)=(AB)+ Applying DeMorgan's theorem,

F = (AB) ° (BC)

which has three NAND forms, as illustrated in Figure 2.11.

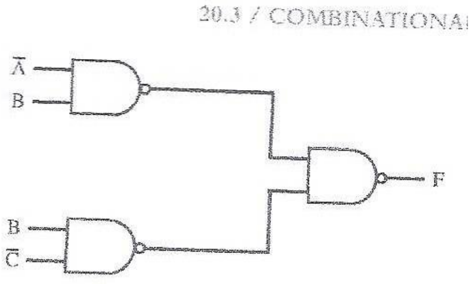


Figure 20.11 NAND Implementation of Table 20.3

The multiplexer connects 1utinle inputs to a single output. At any time, one of the inputs is selected to to the output. A general block diagram representation is shown in Figure This represents a 4-to-1 ***multiplexer.*** There are four input lines, labeled D0. D'. D-, and D3. One of these lines is selected to provide the output signal F. To select one of the four possible inputs, a 2-bit selection code is needed, and this is impiti meneed as two select lines labeled S1 and S2.

An example 4-to--ˉ multiplexer is defined by the truth table in Table 20.7. This is a simplified form of a truth table. Instead of showing all possible combinations of input variables, it shows tt_e output as data from line D0, D1, D2, or D3. Figure 20.13 shows an implementation using AND, OR, and NOT gates. S1 and S2 are connected to the AND gates in such a way that, for any combination of S1 and S2, three of the AND gates will output 0. The fourth AND gate will output the value of the selected line, which is either 0 or 1. Thus, three of the inputs to the OR gate are always 0, and the output of the OR gate will equal the value of the selected input gate. Using this regular organization, it is easy to construct multiplexers of size 8-to-1,16-to-1, and so on.

Multiplexers are used in digital circuits to control signal and data routing. An example is the loading of the program counter (PC). The value to be loaded into the program counter may come from one of several different sources:

A binary counter, if the PC is to be incremented for the next instruction

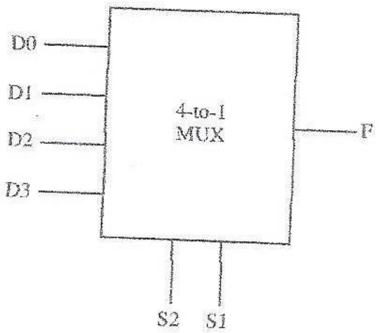
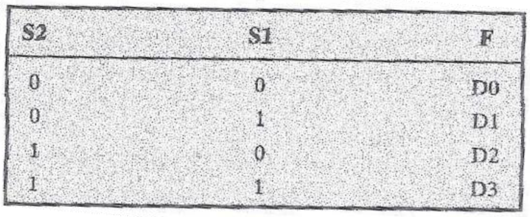


Figure 20.12 4-to-1 Multiplexer Representation

Table 20.7 4-to-1 Multip

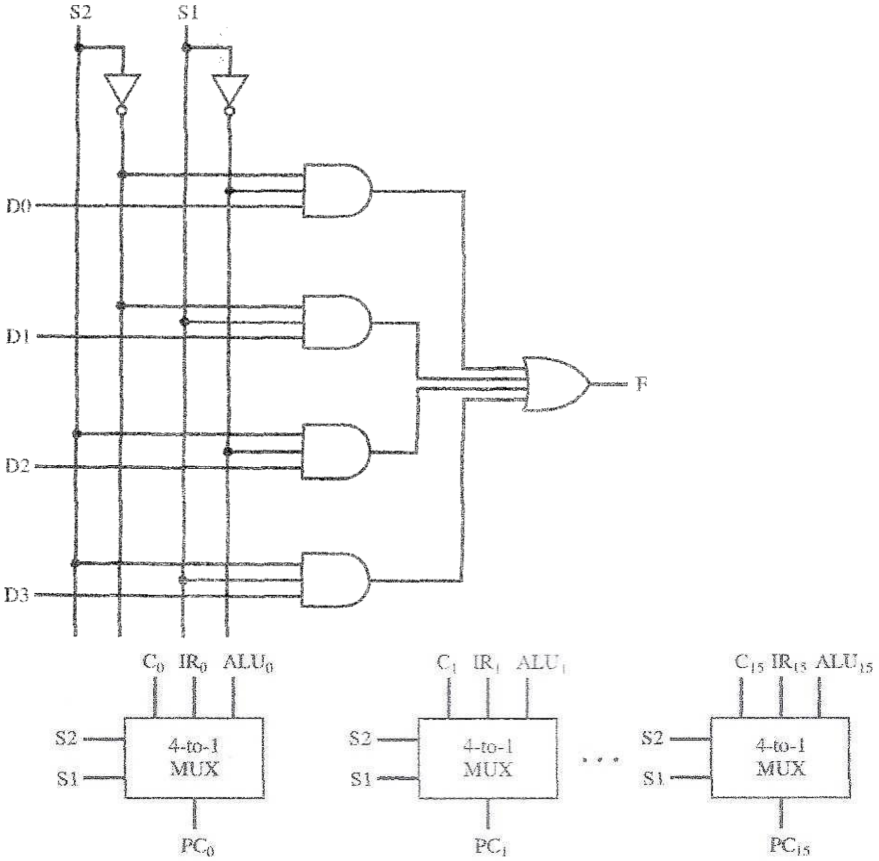


S2	S1	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

- The instruction register, if a branch instruction using a direct address has just been executed
- The output of the AT U, if the branch instruction specifies the address using a displacement mode

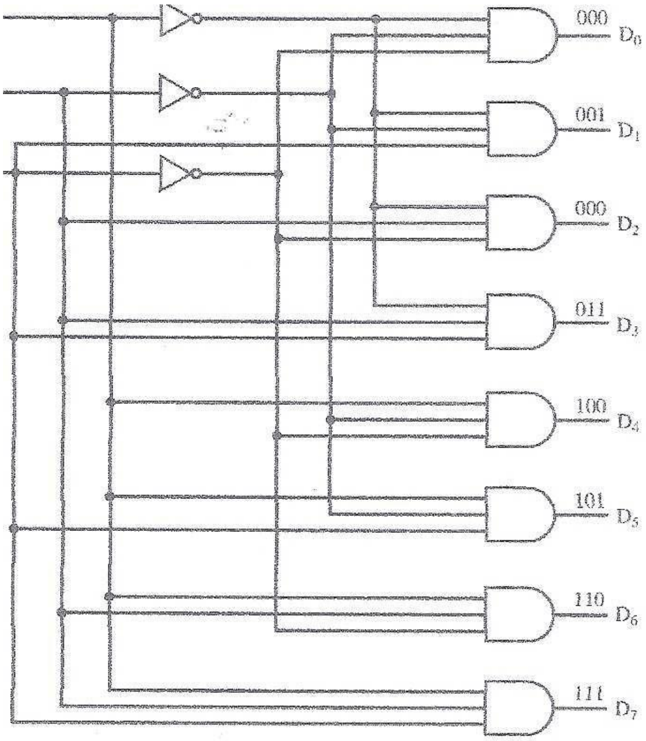
These various inputs could be connected to the input lines of a mulltiplexer, with the PC connected to the output line. The select lines determine which value is loaded into the PC. Because the PC contains multiple bits, multiple multiplexers are used, one per bit. Figure 20.14 illustrates this for 16-bit addresses.

A decoder is a combinational circuit with a number of output lines, only one of which is asserted at any time, dependent on the pattern of input lines. In general, a decoder has n inputs and 2" outputs. Figure 20.15 shows a decoder with three inputs and eight outputs.

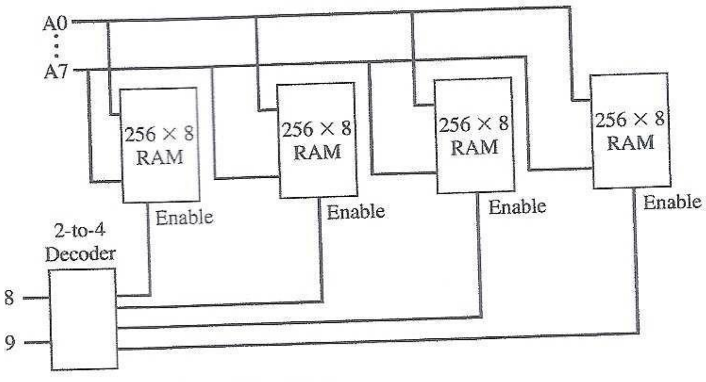


Decoders find many uses in digital computers. One example is address decoding. Suppose we wish to construct a 1K-byte memory using four 256 X 8-bit RAM chips. We want a single unified address space, which can be broken down as follows;

Address	**Chip**
0000-00FF	0
0100-01FF	1
0200-02FF	2
0300-03FF	3



ure 20.15 Decoder with 3 Inputs and $2^3 = 8$ Outputs



igure 20.16 Address Decoding

Each chip requires 8 address lines, and these are supplied by the lower-order 8 bits of the address. The higher-order 2 bits of the 10-bit address are used to select one of the four RAM chips. For this purpose, a 2-to-4 decoder is used whose output enables one of the four chips, as shown in Figure 20.16.

With an additional input line, a decoder can be used as a demultiplexer. The demultiplexer performs the inverse function of a multiplexer; it connects a single input to one of several outputs. This is shown in Figure 20.17. As before, n inputs are decoded to produce a single one of 2" outputs. All of the 2" output lines are ANDed with a data input line. Thus, the *n* inputs act as an address to select a particular output line, and the value on the data input line (0 or 1) is routed to that output line.

The configuration in Figure 20.17 can be viewed in another way. Change the label on the new line from Data *Input* to *Enable. This* allows for the control of the timing of the decoder. The decoded output appears only when the encoded input is *present and the* enable line has a value of 1.

3.3 READ ONLY MEMORY

Combinational circuits are often referred to as "memoryless" circuits, because their output depends only on their current input and no history of prior inputs is retained.

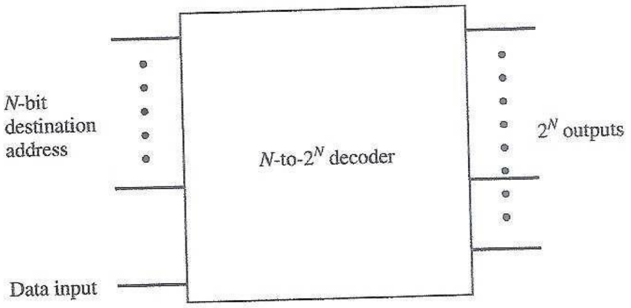
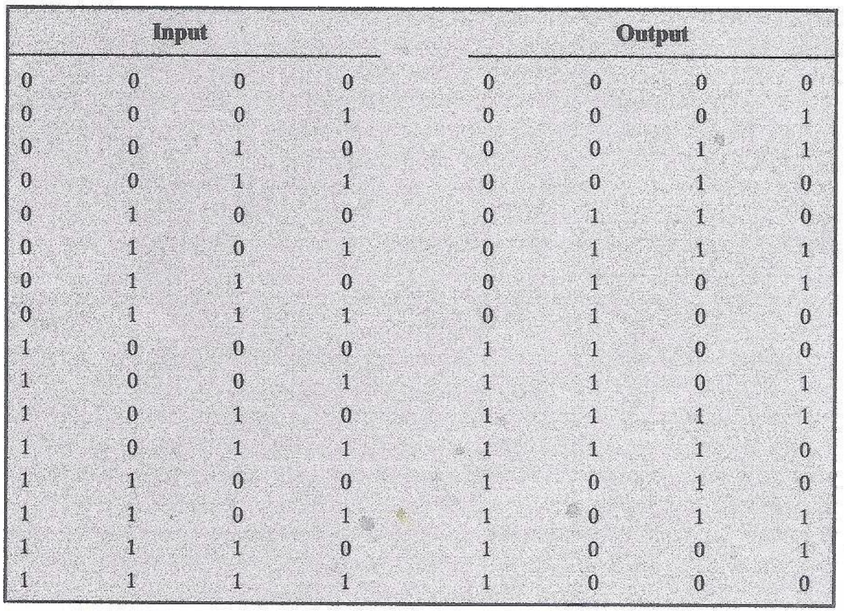


Figure 20.17 Implementation of a Demultiplexer Using a Decorder

Table 20.8 Truth Table for a ROM



Input				Output			
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

However, there is one sort of memory that is implemented with combinational circuits, namely *read-only memory* (ROM).

Recall that a ROM is a memory unit that performs only the read operation. This implies that the binary information stored in a ROM is permanent and was created during the fabrication process. Thus, a given input to the ROM (address lines) always produces the same output (data lines). Because the outputs are a function only of the present inputs, the ROM is in fact a combinational circuit.

A ROM can be implemented with a decoder and a set of OR gates. As an example, consider Table 20.8. This can be viewed as a truth table with four inputs and four outputs. For each of the 16 possible input values, the corresponding set of values of the outputs is shown. It can also be viewed as defining the contents of a 64-bit ROM consisting of 16 words of 4 bits each. The four inputs specify an address, and the four outputs specify the contents of the location specified by the address. Figure 20.18 shows how this memory could be implemented using a 4-to-16 decoder and four OR gates. As with the PLA, a regular organization is used, and the interconnections are made to reflect the desired result.

ADDERS

So far, we have seen how interconnected gates can be used to implement such functions as the routing of signals, decoding, awd ROM. One essential area not yet addressed is that of arithmetic. In this brief ova-view, we will content ourselves with looking at the addition function.

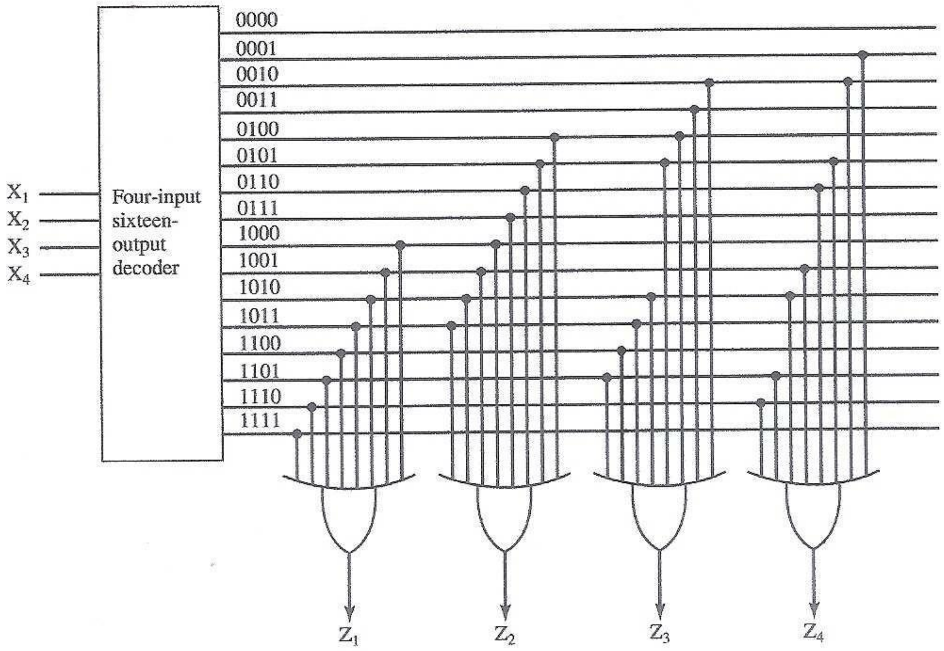
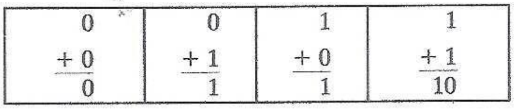


Figure 20.18 A 64-Bit ROM

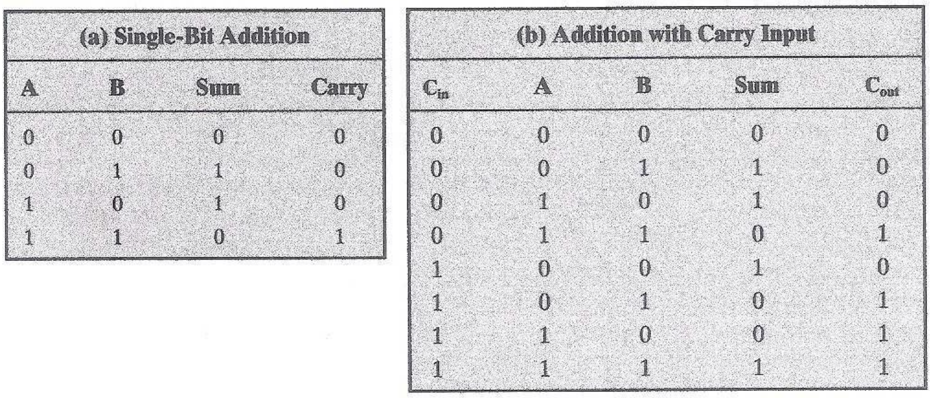
Binary addition differs from Boolean algebra in that the result includes a carry term. Thus,



0 + 0 0	0 + 1 1	1 + 0 1	1 + 1 10

However, addition can still be dealt with in Boolean terms. In Table G0.9a, we show the logic for adding two input bits to produce a 1-bit sum and a carry bit. This truth table

Table 20.9 Binary Addition Truth Tables



(a) Single-Bit Addition

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b) Addition with Carry Input

C_{in}	A	B	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

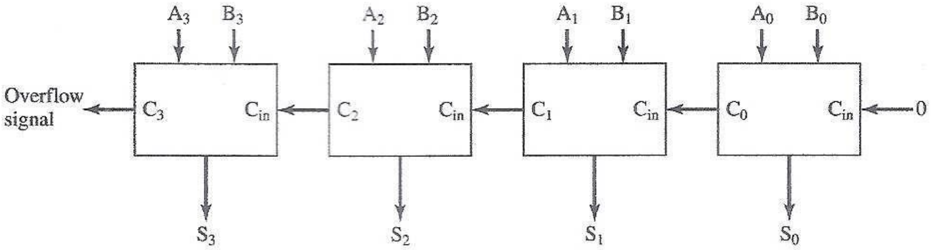


Figure 20.19 4-Bit Adder

could easily be implemented in digital logic However, we are not interested in performing addition on just a single pair of bits. Rather, we wish to add two n-bit numbers. This can be done by putting together a set of adders so that the carry from one adder is provided as input to the next. A 4-bit adder is depicted in Figure 20.14.

For a multiple-bit adder to work, each of the single-bit adders must have three inputs, including the carry from the next-lower-order adder. The revised truth table appears in Table 20.9b. The two outputs can be expressed:

Sum = A BC + ABC + ABC + ABC

Carry=AB+AC+BC

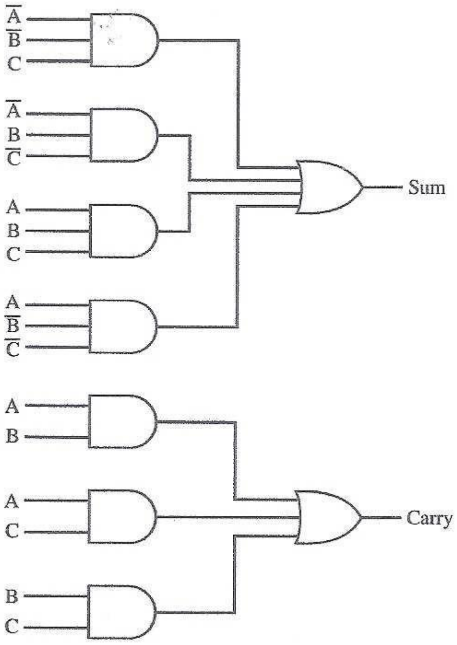
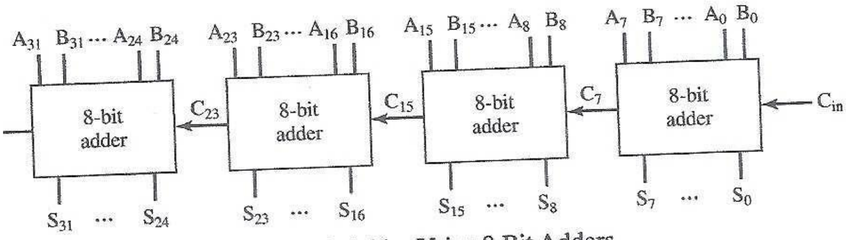


Figure 20.20 Implementation of an Adder



e 20.21 Construction of a 32-Bit Adder Using 8-Bit Adders

Thus we have the necessary logic to implement a multiple-bit adder such as shown in Figure 20.21. Note that because the output from each adder depends on the carry from the previous adder, there is an increasing delay from the least significant to the most significant bit. Each single-bit adder experiences a certain amount of gate delay, and this gate delay accumulates. For larger adders, the accumulated delay can become unacceptably high.

If the carry values could be determined without having to ripple through all the previous stages, then each single-bit adder could function independently, and delay would not accumulate. This can be achieved with an approach known as carry *lookahead.* Let us look again at the 4-bit adder to explain this approach.

We would like to come up with an expression that specifies the carry input to any stage of the adder without reference to previous carry values. We have

$C_o = A_O B_Q$ (20.4)

$C_i = A_j B_j + A_j A_0 B_0 + B_I A_G B_O$ (20.5)

$C_o = A_O B_Q$ (20.4)

$C_i = A_j B_j + A_j A_0 B_0 + B_I A_G B_O$ (20.5)

$C2 = A2B2 + A_2 A_j B_i + A_2 A_j A_0 B_o + A_2 B_j A_0 B_o + B2ACB1 + B_2 A_j A_0 B_o + B2BjA0Bo$

This process can be repeated for arbitrarily long adders. Each carry term can be expressed in SOP form as a function only of the original inputs, with no dependence on the carries. Thus, only two levels of gate delay occur regardless of the length of the adder.

For long numbers, this approach becomes excessively complicated. Evaluating the expression for the most significant bit of an n-bit adder requires an OR gate with n - 1 inputs and n AND gates with from 2 to n + 1 inputs. Accordingly, full carry lookahead is typically done only 4 to 8 bits at a time. Figure 20.21 shows how a 32-bit adder can be constructed out of four 8-bit adders. In this case, the carry must ripple through the four 8-bit adders, but this will be substantially quicker than a ripple through thirty-two 1 bit adders.

4.0 CONCLUSION

The Bolean Algerba is the mathematical foundation of digital logic: which turns out to be convenient tool in analysis and design.

5.0 SUMMARY

Boolean algebra makes use of two variables which may take the value 1 or 0. The operators make use of sign in symbolic form. The postulates defines the way in which Boolean expression are interpreted.

1.0 INTRODUCTION

The fundamental building block of all digital logic circuit is the gate. Logical functions are implemented by the interconnection of gates.

-

3.1 Gate

3.2 basic logic gates

3.1 Implementation of Boolean function(7-9)

3.2 Simplification of Boolean function(9-20)

3.3 Reading memory(20-24)

4.0 CONCLUSIONS

Combination circuit can be defined in three ways and Boolean functions can be simplified in four ways, namely

Algebraic simplification, karnaugh maps and quine.mcklukey tables.

5.0 SUMMARY

In summary, the karnaugh map of simplifying Boolean function appear to be more cumbersome. The quince mckluskey method is best suitable for programming on a computer to give an automatic tools for producing minimized Boolean expressions.

6.0 TUTOR MARKED ASSIGNMENT.

1 Write the Boolean expression for a four input NAND gate.

2. Design an 8 to 1 multiplexer.

7.0 REFERENCE/FURTHER READING

Mand,M., and Kime, C.logic and Computer Design fundamentals. Upper saddle river, NJ: prentice hall, 2004.