KARNATAKA STATE OPEN UNIVERSITY

MUKTHAGANGOTRI, MYSORE- 570 006

DEPARTMENT OF STUDIES IN INFORMATION TECHNOLOGY

M.Sc IN INFORMATION SCIENCE

II SEMESTER

COMPUTER ORGANIZATION

IS 2.1 BLOCK 1 TO 4

IS 2.1
Computer Organization

Preface

This material is prepared to give an overview of Computer Organization for the Second Semester course in M.Sc. (IT) curricula. It is suitable for both hardware and software-oriented students. To study the design details of computer organization and the various concepts related to computer organization, this material has been prepared. The whole material is organized into four modules each with four units. Each unit lists out the objectives of study along with the relevant questions and suggested reading to better understand the concepts.

Module-1: Gives an introduction to computer organization. It starts with components of computer, computer functions, interconnection networks. It also describes main memory operations and addressing modes.

Module-2: Introduces the basic processing unit. It starts with some fundamental concepts. It describes movement of data and instructions execution. It also describes performance considerations, hardwired control and microprogrammed control.

Module-3: Describes the input-output organization. It discusses different types of accessing I/O devices. It also introduces concepts of interrupts, DMA and I/O hardware.

Module-4: Introduces the concept of system memory. It starts with basic memory concepts. It explains semiconductor memory chips. It also explains the mechanism of cache memory. It ends with an interesting concept called virtual memory.

We thank everyone who helped us directly or indirectly in preparing this material. Without their support, this material would not have been a reality.

Karnataka State Open University

Mukthagangothri, Mysore – 570 006

Second Semester M.Sc in Information Science

Data Base Management Systems

UNIT 1: COMPUTER ORGANIZATION AND ARCHITECTURE

Structure

1.0 OBJECTIVES

At the end of this unit you will be able to

- Identify various types of computers
- Identify the functional units of a computer
- State the functions performed by each functional unit
- Discuss the interaction between the CPU and memory
- Understand the difference between Computer organization and architecture

1.1 INTRODUCTION

This unit is about the basic computer organization. It describes what a computer is, its types, its functional components, and how does these functional components work together as a system? It also gives you an overview of the basic operational concepts of computers which involves the functional components such as input, output, memory, arithmetic-logic unit and control unit. We understand the difference between Computer

organization and architecture. The subsequent modules will discuss in detail the organization of each component of the system.

1.2 TYPES OF COMPUTERS

A computer is an electronic device that is capable of accepting data, process the accepted data according to the given sequence of instructions, storing data, presenting data according to given format and communicating data over networks.

There are two broad classes of computers based on the type of input data they accept. They are: Analog Computers and Digital computers.

Analog Computers: Electronic devices that are capable of accepting data in analog or time varying form for processing.

Digital computes: Electronic devices that are capable of accepting data in the digital form for processing. These computers process the accepted data according to a given sequence of instructions known as a program. The result of processing data is information. The programs and the result reside in the internal storage called computer memory.

Today, there are many types of computers that differ in size, cost, computational power and its indented use. They are:

- Desktops computers
- Portable notebook computers
- Work stations
- Mainframes
- Super computers

Desktops Computers:

A desktop computer is a personal computer that is designed to be accommodated conveniently on top of a typical office desk. A desktop computer typically consists of various units such as the processor, the display monitor and input devices - usually a keyboard and a mouse that are connected together during installation. Today, almost all desktop computers include a built-in modem, a CD-ROM drive, a multi-gigabyte

magnetic storage drive. In businesses and at home, most desktop computer users can share resources such as printers, plotters and fax machines by getting connected to a local area network.

Portable notebook computers:

A portable notebook computer is a compact version of a personal computer with all components of a PC packaged into a single unit which is handy and portable. Laptop is an example for this type of computer.

Work stations:

A work station is a high computational powered personal computer having high resolution graphics terminals and improved input-output capabilities. It often finds its use in engineering applications and interactive graphics applications.

Mainframes:

A Main frame computer is a large data processing system used in medium and large sized business units. It is implemented using two or more central processing units and designed to operate at very high speeds for large volumes of data. A Mainframe is also known as an Enterprise system.

Super Computers:

A super computer is a high-performance computing device meant for highly calculation-intensive tasks involving problems on quantum physics, weather forecasting, climate research, molecular modeling and physical simulations such as a rocket design or a submarine design etc.

1.3 FUNCTIONAL UNITS

A computer in its simplest form consists of five components. They are: input, output, memory, arithmetic and logic unit, and control unit as shown in the Figure 1.1.

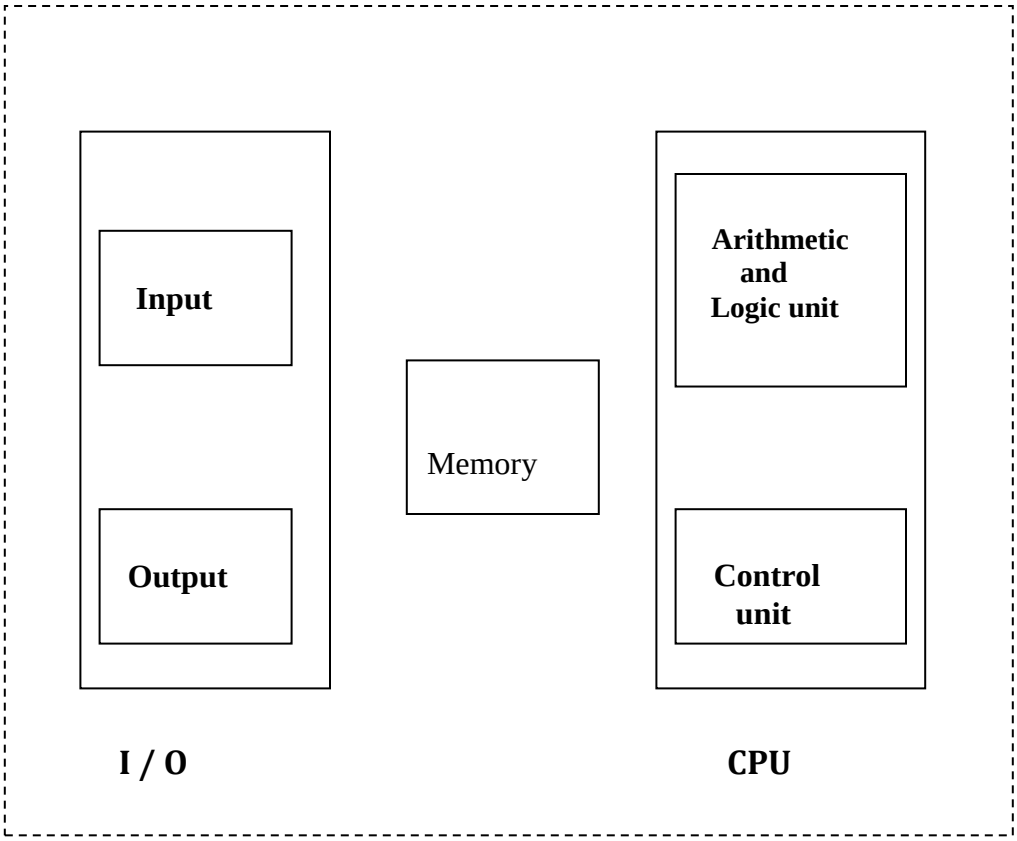


Figure 1.1 Functional units of a computer

The operations performed by a computer using the functional units can be summarized as follows:

- It accepts information (program and data) through input unit and transfers it to the memory
- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit for processing
- Processed information leaves the computer through an output unit
- The control unit controls all activities taking place inside a computer.

1.4 BASIC OPERATIONAL CONCEPTS

A program is nothing but sequence of instructions that instruct the computer to perform some specified operation on given data. Programs reside in the main memory of the

computer. In-order to perform the specified operations, the instructions are brought from memory into the processor. The Figure 1.2 describes the connections between the CPU and the main memory. The processor contains arithmetic and logic unit as the main processing unit, the control unit to control and coordinate all activities in the system. It also contains a number of registers used for various purposes (e.g., temporary storage of data), such as the Instruction Register (IR), the Program Counter (PC), the Memory Address Register (MAR), Memory Data Register (MDR), and the general-purpose registers.

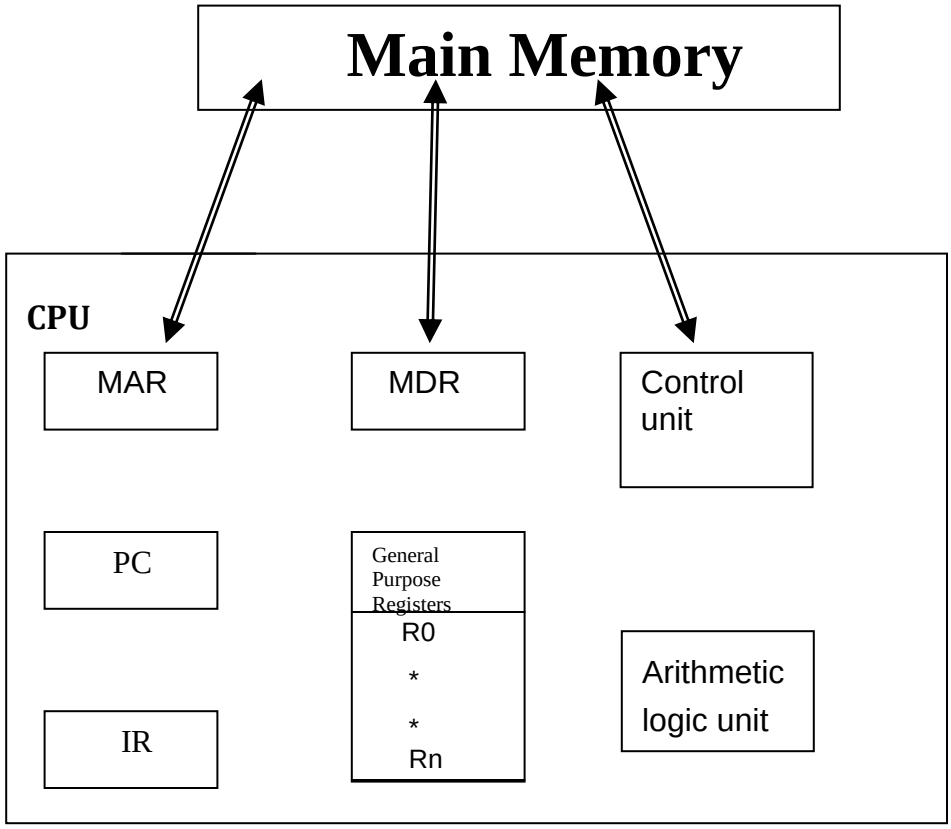


Figure 1.2 Connections between the CPU and the main memory.

Instruction Register (IR): contains the instruction that is currently being executed. Its output is available to the control circuit that generates the timing signals for control of the actual processing circuit needed to execute the instruction.

Program Counter (PC): is a register that contains the memory address of the instruction currently being executed. During the execution of the current instruction, the content of program counter is updated to correspond to the address of the next instruction.

Memory Address Register (MAR): holds the address of the memory location to or from which data is to be transferred.

Memory Data Register (MDR): contains the data to be written into or read-out of the addressed memory location.

General-purpose Registers: are used for holding data, intermediate results of operations. They are also known as **scratch-pad registers.**

Let us consider some typical operating steps involving instruction fetch and execution:

INSTRUCTION FETCH

- Program gets into the memory through an input device
- Execution of a program starts by setting the PC to point to the first instruction of the program.
- The contents of PC are transferred to the MAR and a Read control signal is sent to the memory
- The addressed word (here it is the first instruction of the program) is read out of memory and loaded into the MDR
- The contents of MDR are transferred to the IR for instruction decoding

INSTRUCTION EXECUTION

- The operation field of the instruction in IR is examined to determine the type of operation to be performed by the ALU
- The specified operation is performed by obtaining the operand(s) from the memory locations or from GP registers.
    - Fetching the operands from the memory requires sending the memory location address to the MAR and initiating a Read cycle.
    - The operand is read from the memory into the MDR and then from MDR to the ALU.
    - The ALU performs the desired operation on one or more operands

fetched in this manner and sends the result either to memory location or to a GP register.

- The result is sent to MDR and the address of the location where the result is to be stored is sent to MAR and Write cycle is initiated.

Thus, the execute cycle ends for the current instruction and the PC is incremented to point to the next instruction for a new fetch cycle.

1.5 DEFINITION OF COMPUTER ORGANIZATION AND ARCHITECTURE

Computer architecture refers to those attributes of a system visible to a programmer, or put another way, those attributes that have a direct impact on the logical execution of a program. Computer organization refers to the operational units and their interconnection that realize the architecture specification. For examples, architecture attributes include the instruction set, the number of bits to represent various data I/O mechanisms, and technique for addressing memory. Whereas, organization attributes include those hardware details transparent to the programmer, such as control signals, interfaces between the computer and peripherals, and the memory technology used.

In computer science and engineering, computer architecture is the practical art of defining the structure and relationship of the subcomponents of a computer. As in designing the architecture of buildings, *architecture* can comprise many levels of information. Computer architecture is primarily logical, positing a conceptual system that serves a particular purpose. Computer organization helps optimize performance-based products.

Differences between computer organization and architecture:

Computer organization is how operational attributes are linked together and contribute to realize the architectural specifications. Computer architecture is the architectural attributes like physical address memory, CPU and how they should be made and made to coordinate with each other keeping the future demands and goals in mind.

A computer's architecture is its abstract model and is the programmer's view in terms of instructions, addressing modes and registers. A computer's organization expresses the realization of the architecture. Architecture describes what the computer does and organization describes how it does it.

Check your progress

1. Define a computer. How is it classified based on its input?

2. What are the types of computers?

3. Explain the important operations performed by a computer using the functional units.

4. What is the function of a program counter and an instruction register?

5. What is the difference between computer organization and architecture?

1.6 SUMMARY

A computer is said to be operational as a system, when its functional units are interconnected by a group of wires called Bus**.** One can choose his computer system based on his requirements. And a comfortable choice can be made depending on one's needs related to the cost or speed. In this unit we discussed about functional units of computer, basic operational concepts, definition of organization and architecture.

1.7 KEYWORDS

Computer: A computer is a programmable device that receives input, stores and manipulates data, communicates data and provides output in a suitable format.
Instruction: An explicit command given to a computer.

Register: A high speed storage element.

Program: A sequence of instructions stored in memory and processed by a processor.

Analog: data in analog or time varying form for processing.

Digital: data in digital or time invariant form for processing.

1.8 ANSWERS TO CHECK YOUR PROGRESS

1. 1.2
2. 1.2
3. 1.3
4. 1.4
5. 1.5

1.9 UNIT-END EXERCISES AND ANSWERS

1. How are computers classified based on their size, cost, computational power and indented use?
2. Name the functional units of a computer and describe its functions.
3. How does the CPU interact with the main memory? Explain with necessary block diagram.
4. Differentiate between computer organization and architecture.

Answers: SEE

1. 1.2
2. 1.3
3. 1.4
4. 1.5

1.10 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002.
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006.
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 2: COMPUTER STRUCTURES

Structure

2.0 OBJECTIVES

At the end of this unit you will be able to

- Computer functions
- Define a Bus
- Discuss the various Bus structures
- Memory arrangement
- Addressing and Encoding of information

2.1 INTRODUCTION

This unit is about the computer structures. It describes Computer function, their Interconnection structures. It defines Bus interconnection. It shows how various parts of a computer are interconnected with bus structure. It also deals with Memory locations, Addressing and Encoding of information.

2.2 COMPUTER FUNCTIONS

The main function of a computer is to run programs. The computers are used today for an almost unlimited range of applications. However, irrespective of the application for which a computer is used we can identify a few basic functions that are performed by all computers. All the computer applications make use of these basic functions of computers in different ways and combinations. There are basically four basic functions of computers. They are input, storage, processing and output. These are described below:

1. INPUT: This is receiving or accepting information from outside sources. We input data and instructions through input devices which are keyboard, mouse, scanner, etc. The most common way of performing input function is through the information entered through the keyboard and the click of mouse. Of course, there are many other types of devices for receiving such information - for example, the web cam. Computers are also able to receive information stored in other devices like DVD disks and pen drives. Computers are also able to receive information from other computers and similar devices. When we use computers for automatic control of machines and processes, computers can also receive information directly from such equipments and processes.

2. PROCESSING: The computer processes data. This is really the core of computer operation. The computer processes the data that is fed to the computer by various means and the data already contained in internal memory to produce the results that is the core of all computer applications, which is done by the Central Processing Unit (CPU).

3. OUTPUT: After processing the data the computer gives the result as an output. Output devices are the monitor (in the case of visual output), speakers (in the case of audio output), printers, etc. The results of the processing are made available for use by any user or other devices. When a computer is connected to other devices,

including through Internet, this output is in the form of electrical pulses. The output data can also be recorded on to an external recording medium such as a DVD disk.

4. STORAGE-: We can save our data for future use in the computer itself. There are several storage devices also like removable disks, CDs, etc. The information in the computer is stored in computer in several different ways depending on how the information is used. For simplicity we will classify in two broad categories. First is the memory in the central processing unit of the computer, and second is the auxiliary memory. The auxiliary memory includes devices such as fixed hard drives.

2.3 INTERCONNECTION STRUCTURES

A Computer consists of a set of components (CPU, I/O, memory) that communicates with each other. Collection of paths connecting various computer components is known as Interconnection Structures. The design of this structure will depend on the exchange that must be made between modules. The various types of exchanges/transfers are: Memory to CPU, CPU to Memory, I/O to CPU, CPU to I/O, I/O to or from Memory (Direct Memory Access (DMA)).

2.4 BUS INTERCONNECTION

Bus: A Bus is a collection of wires or distinct lines meant to carry data, address and control information. The functional components of a computer must be connected in-order to make a system operational. The connections can be made in several ways using a Bus.

- **Data Bus**: it is used for transmission of data. The number of data lines corresponds to the number of bits in a word.
- **Address Bus:** it carries the address of the main memory location from where the data can be accessed.

- **Control Bus:** it is used to indicate the direction of data transfer and to coordinate the timing of events during the transfer.

The different functional units of a computer can be connected through a bus structure such as:

- A Single-bus structure
- A Two-bus structure

SINGLE-BUS STRUCTURE:

All units are connected to a single bus as shown in Figure 2.1. The bus can be used for only one transfer at a time since only two units can actively use the bus at any given instant of time. When multiple requests arise for the use of bus, then the Bus control lines are used for managing it. The primary advantage of this structure is its low cost and flexibility for attaching peripherals. But the drawback is its low operating speed. This type of structure is mainly found in small computers such as minicomputers and microcomputers.

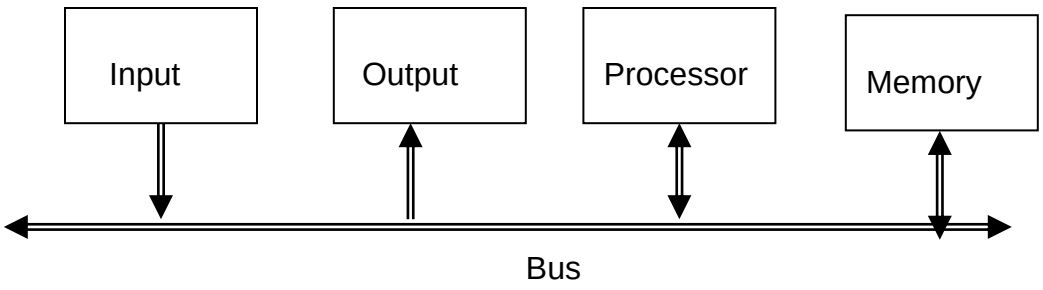


Figure 2.1 Single-bus Structure

TWO-BUS STRUCTURE:

The bus is said to perform two distinct functions by connecting the I/O units with memory and processor unit with memory. The processor interacts with the memory through a memory bus and handles input/output functions over I/O bus. The I/O transfers are always under the direct control of the processor, which initiates transfer and monitors their progress until completion. The main advantage of this structure is good operating speed but on account of more cost.

The Figure 2.2 shows the two-bus structure and Figure 2.3 shows the alternate arrangement of two-bus structure.

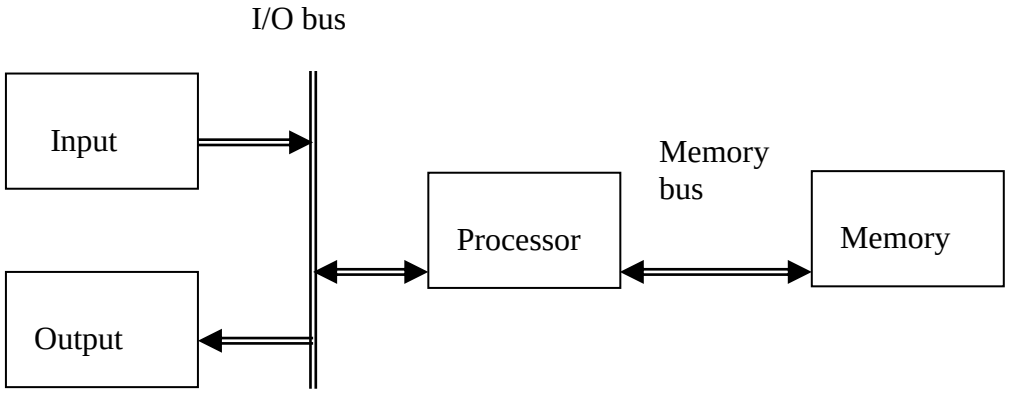


Figure 2.2 Two-bus Structure

AN ALTERNATIVE TWO-BUS STRUCTURE:

Here the positions of memory and processor are interchanged. I/O transfers are directly made to or from the memory. So, special purpose processor called peripheral processor is used for providing the necessary controls over the actual data transfer.

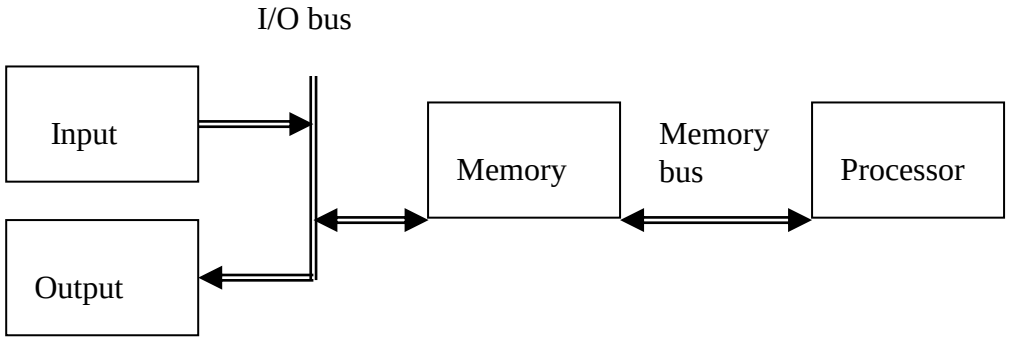


Figure 2.3: An alternative Two-bus Structure

2.5 MEMORY LOCATIONS, ADDRESSES, AND INFORMATION ENCODING

We now discuss the way the programs are executed in a computer from the programmer's point of view. Number and character operands, as well as instructions, are stored in the memory of a computer.

The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, the bits are normally not handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of m bits can be stored or retrieved in a single, basic operation. Each group of m bits is referred to as a word of information, and m is called the word length. The memory of a computer can be schematically represented as a collection of words as shown in Figure 2.4. Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit 2's complement number or four ASCII characters, each occupying 8 bits. A unit of 8 bits is called a byte. Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct addresses for each item location. It is customary to use numbers from 0 through $2^n - 1$, for some suitable values of n, as the addresses of successive locations in the memory. The 2^n addresses constitute the address space of the computer, and the memory can have up to 2^n addressable locations. A 32-bit address creates an address space of 2^{32} or 4G (4 Giga) locations.

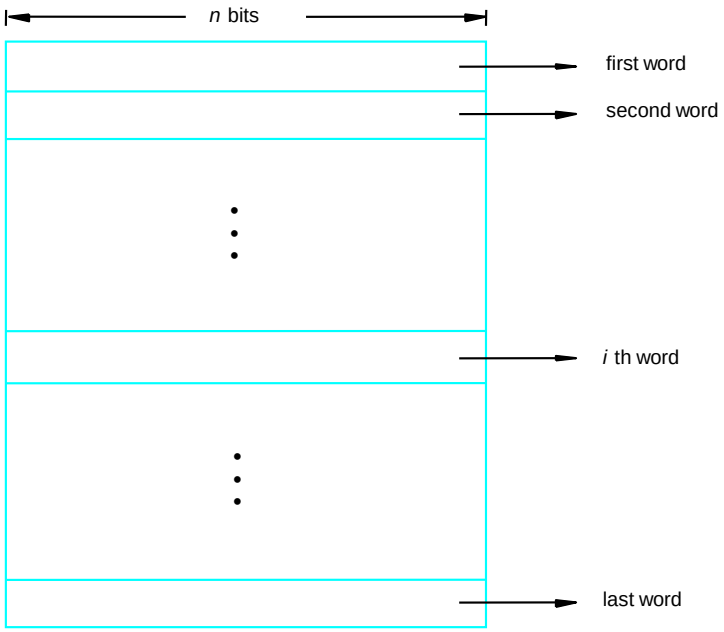


Figure 2.4: Main Memory Addresses

BYTE ADDRESSABILITY:

We now have three basic information quantities to deal with: the bit, byte and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. The most practical assignment is to have successive addresses refer to successive byte. Contents of the memory locations can represent either instructions or operands. Operands can be either numbers or characters.

Representation of Numbers in main memory:

Consider a 32 bit pattern to represent a signed integer. Figure 2.5 shows how a number can be stored in a 32 bit word.

32 bits

b_{31}	b_{30}	. . .	b_1	b_0

Sign bit: $\mathbf{b_{31} = 0}$ for positive numbers

$\mathbf{b_{31} = 1}$ for negative numbers

Magnitude = $b_{30} . 2^{30} + b_{29} . 2^{29} + ... + b_1 . 2^1 + b_0 . 2^0$

Figure 2.5: A signed integer

Magnitude can range from 0 to $2^{31} – 1$ and the numbers are said to be in binary positional notation. The above encoding format is called Signed Magnitude representation. The other two binary representations are 1's compliment and 2's compliment representations. Representation of positive numbers is the same in the three cases. The difference is only in the negative number. In all the three methods the left most bit is the sign bit (i.e.0 represents positive number and 1 represents negative number). 2's compliment method is the most suitable one and is used in all modern computers.

Representation of Characters in main memory

Characters can be letters of the alphabet, decimal digits, punctuation marks etc. They are represented by codes that are usually 6 – 8 bits long. Figure 2.6 shows how 4 characters in ASCII can be stored in a 32 bit word.

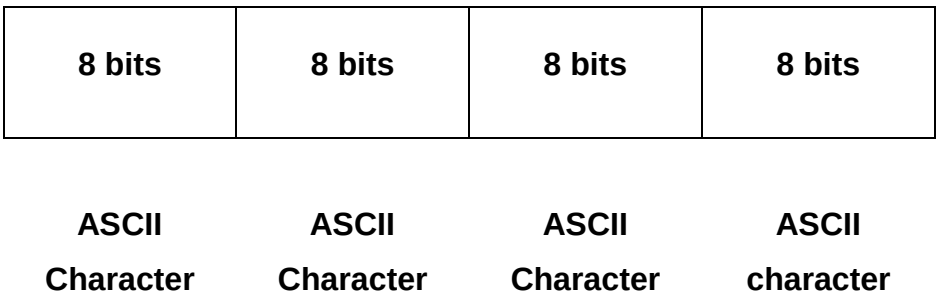


Figure: 2.6: A character

Representation of Instructions in main memory

A main memory word can also be used to represent an instruction. One part of the word specifies the operation to be performed and the other part specifies operand address. Each of these parts is called as 'field', represented as shown in Figure 2.7

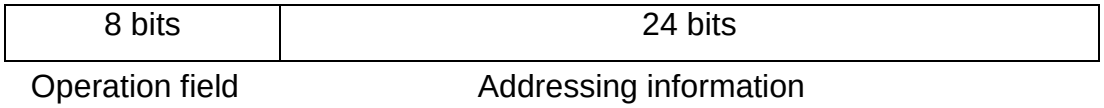


Fig 2.7: An instruction

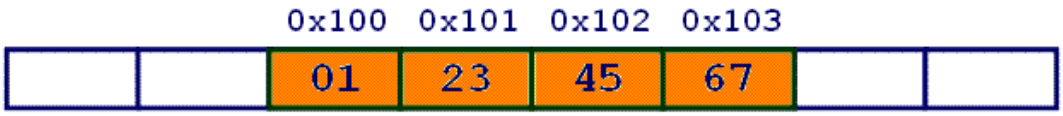
Here the 8 bit operation field can encode 2^8 (256) distinct instructions.

Addressing information is given in a variety of ways. The different ways in which operands can be named in machine instructions are called addressing modes. Memory words whose addresses are specified by the instructions are interpreted as operands. Whether an operand is a character or a numeric data item is determined by the operation field of the instruction. An operand may be either shorter or longer than one word. An operand length of 8 bit is convenient, because this size is used to encode character data.

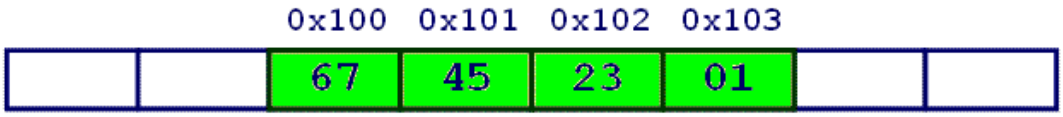
BIG-ENDIAN AND LITTLE ENDIAN ASSIGNMENTS

Little and big-endian are two ways of storing multibyte data-types (int, float, etc). In little-endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big-endian machines, first byte of binary representation of the multibyte data-type is stored last.

Suppose integer is stored as 4 bytes (assuming integer is 2 bytes) then a variable x with value 0×01234567 will be stored as following.



Big Endian



Little Endian

Check your progress

1. What are functions of a computer?

2. What are interconnection structures?

3. Explain Bus interconnection structure.

4. Explain the structure of main memory.

5. What is big-endian and little-endian?

SUMMARY

This unit dealt with the computer structures. It described Computer functions, their Interconnection structures. It also defined Bus interconnection. It also dealt with Memory locations, Addressing and Encoding of information.

.

2.6 KEYWORDS

Interconnection Structures: Collection of paths connecting various computer components.

Bus: A collection of wires or distinct lines meant to carry data, address and control information.

Addressing mode: The different ways, in which operands can be named in machine instructions.

2.7 ANSWERS TO CHECK YOUR PROGRESS

1 2.2
2 2.3
3 2.4
4 2.5
5 2.5

2.8 UNIT-END EXERCISES AND ANSWERS

5. What are different types of buses?
6. Explain various types of Bus interconnection.
7. Describe the structure of main memory.
8. With example, distinguish between big-endian and little-endian.

Answers: SEE

1. 2.4
2. 2.4
3. 2.5
4. 2.5

2.9 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002.
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006.
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 3: Main Memory Operations

Structure

3.0 OBJECTIVES

At the end of this unit you will be able to

- Identify main memory operations
- Explain instructions fetch and instructions execution.
- Identify various instruction formats.
- Understand different addressing techniques.

3.1 INTRODUCTION

After knowing the general concepts of computer organization and the way computer operates, in this unit, we will see the main memory operations, various types of instructions and instructions sequencing. It also deals with instruction execution and straight-line sequencing, condition codes.

3.2 MAIN MEMORY OPERATIONS

To execute an instruction the instructions must be transferred from the main memory to the CPU.

This is done by the CPU control circuits. Operands and results must also be moved between the main memory and the CPU. Thus two basic operations involving the main memory are needed namely, **Load (or Fetch or Read) and Store (or Write).**

Load operation: Transfers a copy of the contents of a specific memory location to the CPU. The Word in the main memory remains unchanged. To start a Load or Fetch operation, CPU sends the address of the desired location to the main memory and requests to read its contents. The main memory reads the data stored at that address and sends them to the CPU.

Store operation: Transfers a word of information from the CPU to a specific main memory location, destroying the former contents of that location. Here the CPU sends the address of the desired location to the main memory, together with the data to be written to that location.

3.3 INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations

1. Data transfers between the main memory and the CPU registers
2. Arithmetic and logic operations on data
3. Program sequencing and control
4. I/O transfers

Notations used:-

a) Register Transfer Notation (RTN):- Possible locations involved in transfer of information are memory location, CPU registers or registers in the I/O subsystem. We

identify the names for the addresses of memory location as LOC, PLACE, A, VAR2 etc and the names for CPU registers as R0, R5 etc. The contents of a location or a register are denoted by placing the corresponding name between square brackets.
E.g. i) R1 _ [LOC] means that the contents of memory location LOC are transferred into register R1.
ii) R3 _ [R1] + [R2] adds the contents of registers R1 and R2 and then places their sum into register R3.

b) Assembly Language Notation:- The same operations can be represented in assembly language format as shown below.
E.g. i) Move LOC, R1
ii) Add R1,R2,R3

BASIC INSTRUCTION TYPES

There are five types of instruction formats in a computer that are commonly used, namely:

1. Three-address instruction format
2. Two-address instruction format
3. One-address instruction format
4. Zero-address instruction format
5. One-and-half address instruction format

Three-address instruction

C = A + B is a high level instruction to add the values of the two variables A and B and to assign the sum to a third variable C. When this statement is compiled, each of these variables is assigned to a location in the memory. The contents of these locations represent the values of the three variables. Hence the above instruction requires the action:

C _ [A] + [B]

To carry out this instruction, the contents of the memory locations A and B are fetched from the main memory and transferred into the processor – sum is computed – result is

sent back to memory and stored in location C. The same action is performed by a single machine instruction (three address instruction)

Add A,B,C

Operands A and B are called the *source operands,* C is called the destination operand, and Add is the operation to be performed on the operands. The general format is

Operation Source1,Source2, Destination

If *k* bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain 3*k* bits for addressing purposes + the bits needed to denote the Add operation.

Two-address instruction

An alternative method is to use two address instruction of the form

Operation Source,Destination

E.g. Add A,B which perform the operation

B _ [A] + [B] Here the sum is calculated and the result is stored in location B replacing the original contents of this location. i.e. operand B acts as source as well as destination. In the former case (three address instruction) the contents of A and B were not destroyed. But here the contents of B are destroyed. This problem is solved by using another two-address instruction to copy the contents of one memory location into another location.

Now C _ [A] + [B] is equivalent to

Move B,C

Add A,C

Note: In all the above instructions, the source operands are specified first, followed by the destination. But there are many computers in which the order is reversed.

One-address instruction

Instead of mentioning the second operand, it is understood to be in a unique location. A processor register usually called the **Accumulator** is used for this purpose.

E.g. i) Add A means that the contents of the memory location A is added to the contents of accumulator and places the sum in the accumulator.

ii) Load A copies the contents of memory location A into accumulator

iii) Store A copies the contents of accumulator to the location A

Depending on the instruction, the operand may be source or destination.

Now the operation C _ [A] + [B] can be performed by executing the following instructions

Load A

Add B

Store C

The above mentioned instructions can also be handled by using general purpose registers. Let R*i* represent a general purpose register.

Move A,R*i*

Move R*i*,A

Add A,R*i*

They are generalizations of Load, Store and Add instructions of the single accumulator case in which register R*i* performs the functions of accumulator.

When a processor has several general-purpose registers, then many instructions involve only operands that are in registers.

E.g., Add R*i*, R*j*

Add R*i*, R*j*, R*k*

In the first instruction, R*j* acts as both source and destination. In the second instruction, R*i*

And R*j* are source and R*k* is the destination.

Advantages of using CPU registers:

i) Data access from these registers is faster than that of main memory locations; because these registers are inside the processor.

ii) Only few bits are needed to specify the register; because the number of registers is very less.

For example, only 5 bits are needed to specify 32 registers.

iii) Instructions, where only register names are contained, will normally fit into one word of memory.

Zero-address instruction

Here locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a *pushdown stack*. A stack is a list of data elements, usually words or bytes, in which these elements can be added or removed through the top of the stack by following the LIFO (last-in-first-out) storage mechanism. A processor register called **stack pointer**. **(SP)** is used to keep track of the address of the element at the top of the stack at any given time. The terms **push** and **pop** are used to describe placing a new item on the stack and removing the top item from the stack respectively.

One and half-address instruction

An instruction that specifies one operand in memory and one operand in a CPU register is referred to as one-and-half address instruction. Using registers it is possible to increase the speed of processing.

3.4 INSTRUCTIONS EXECUTION AND STRAIGHT LINE SEQUENCING

3.4.1 STRAIGHT LINE SEQUENCING

Let us take the operation C ← [A] + [B]. The Example 3.1 shows the program segment as it appears in the main memory of a computer that has a two-address instruction format and a number of general purpose CPU registers.

Example 3.1 A program for C ← [A] + [B]

	Address	Contents
Begin execution here →	*i*	Move A,R0 ; A, B, A are memory locations

i+1	Add B, R0
i+2	Move R0, C
	:
	:
A	data
	:
	:
B	data
	:
	:
C	data

For executing this program, the following steps are to be performed.

1. CPU contains the register called PC which holds the address of the instruction to be executed next. To begin execution, the address of the first instruction 'i' must be placed in PC.

2. CPU control circuits use the information in the PC to fetch and execute the instructions one at a time in the increasing order of addresses. This is called straight line sequencing.

3. As each instruction is executed, the PC is incremented by 4 to point to the next instruction.

Executing a given instruction is a two-phase procedure.

First Phase - Instruction Fetch: Instruction is fetched from the main memory location whose address is in the PC and is placed in the Instruction Register (IR)

Second Phase – Instruction Execute: Instruction in the IR is examined to determine which operation is to be performed. The specified operation is performed by the processor. This may involve fetching operands from main memory (or processor registers), performing an arithmetic or logic operation and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. After the execution phase is over, new instruction fetch can begin.

3.4.2 BRANCHING

Consider the task of adding 'n' numbers. Let the address of memory locations containing n numbers are NUM1, NUM2,…NUM*n*. Separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in the memory location SUM.

Example 3.2 A straight-line program for adding *n* numbers

	Address	Contents
Begin execution here →	*i*	Move NUM1,R0
	i+1	Add NUM2, R0
	i+2	Add NUm3, R0
		:
		:
	i+n-1	Add NUMn, R0
	i+n	Move R0, SUM
		:
		:
	SUM	data
	NUM1	data
	NUM2	data

:

NUMn data

Instead of using long list of Add instruction, it is possible to place a single Ad instruction in a loop as shown in Example 3.3 This loop causes a straight line sequence of instructions to be executed repeatedly. The loop starts at location LOOP and ends at the instruction Branch>0. During each pass through this loop, the address of the next entry is determined and that entry is fetched and added to R0. Assume that the number of entries in the list '*n*' is stored in location N as shown.

Register R1 is used as a counter to determine the number of time the loop is to be executed. Hence the contents of the location N are loaded in register R1 at the beginning of the program. Then within the body of the loop the instruction
Decrement R1 reduces the contents of R1 by 1 each time through the loop. This means that execution of the loop must be repeated as long as the result of the decrement operation is greater than 0.

Example 3.3 Using a loop to add *n* numbers.

Address	Contents
	Move N,R1
	Clear R0
Loop:	*Determine address of "Next" number and add "Next" number to R0*
	Decrement R1
	Branch > 0 Loop
	Move R0,SUM
	:
	:
	Add NUMn, R0

```
                    Move R0, SUM
                          :
                          :
SUM
N           n
NUM1        data
NUM2        data
                          :
NUMn        data
```

We now introduce Branch instruction. This type of instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address. A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way and the next instruction in sequential address order is fetched and executed.

3.5 CONDITION CODES

The processor keeps track of some information about the results of various operations for use by subsequent conditional branch instructions. This is done by recording the required information into individual bits called as ***condition code flags***. In some processors, these flags are grouped together in a special register called the ***condition code register*** or ***status register***.

Four commonly used flags are:-

N (negative) Sets to 1 if the result is negative; otherwise, cleared to 0

Z (zero) Sets to 1 if the result is 0; otherwise, cleared to 0

V (overflow) Sets to 1 if arithmetic overflow occurs; otherwise, cleared to 0

C (carry) Sets to 1 if carry-out results from the operation; otherwise, cleared to 0

Check your progress:

1. Explain the memory operations.
2. Explain the different types of instruction format that are commonly used in a computer.

3.6 SUMMARY

This unit introduced memory operations, various instructions, its representation in different formats. The principles of instructions execution were emphasized. With the understating of this, a reader can do detailed study of instruction execution of any computer architecture.

3.7 KEYWORDS

Instruction: It is basic step to instruct a computer to carry out. A program consists of a sequence of these steps.

.

3.8 ANSWERS TO CHECK YOUR PROGRESS

1. 3.2
2. 3.3

3.9 UNIT-END EXERCISES AND ANSWERS

9. Explain the classification of instructions.
10. What is zero-address and one-and-half address instruction format? Explain their uses.

11. Explain the phases of an instruction execution.
12. Name the flags used with condition codes.

Answers: SEE

1. 3.3
2. 3.3
3. 3.4
4. 3.5

3.10 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002.
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006.
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 4: ADDRESSING MODES AND ASSEMBLY LANGUAGE

Structure

4.0 OBJECTIVES

At the end of this unit you will be able to

- Identify the definition of addressing mode
- Examine and use different addressing techniques
- define what is an assembler, a source program, an object program
- explain what is an assembly language and its the importance in programming
- explain assembly language programs and the manner in which it gets executed
- use and appreciate binary number notation in assembly language programming
- Stacks and queues
- Number representation and operations

4.1 INTRODUCTION

This unit is about the addressing modes and assembly languages. Machine instructions are represented by strings of 0's and 1's. Such patterns of 0's and 1's become

cumbersome while programs are discussed or written. So, these patterns are represented and replaced by symbolic names. Some examples of symbolic names include Move, Add, Branch, Increment etc. When writing programs for a specific computer, the symbolic names are replaced by acronyms called mnemonics. Examples of mnemonics include MOV, ADD, BR, INC etc. Using mnemonics and Register Transfer Notation, a program is written governing the rules. Such a program is called as an assembly language. The set of rules for using the mnemonics in the specification of complete instructions and programs is called the syntax of the language.

4.2 ADDRESSING MODES

The term addressing mode refers to the way in which the operand of an instruction is specified.

1. *Register mode:* The operand is the contents of a CPU register; the name of register is given in the instruction

E.g. Move R1,R2 The contents of R1 is transferred to R2

2. Absolute mode (Direct mode): The operand is in a memory location. The address of the memory location is explicitly given in the instruction.

E.g. Add A,B The contents of the memory location A is added to the contents of the memory location B. The addresses of A and B are given in the instruction itself.

3. Immediate mode: The operand is given explicitly in the instruction. This mode is used in specifying address and data constants in programs

E. g. Move #200, R0

This instruction places the value 200 in register R0.

4. Indirect mode: Here the instruction does not give the operand or its address explicitly. Instead it provides the effective address of the operand. Effective address of the operand is the contents of a register or main memory location, whose address appears in the instruction. We denote indirection by placing the name of the register or the memory address given in the instruction in parenthesis

Let us consider two cases:

i) Add (A), R0

ii) Add (R1), R0

In the first case, when the instruction is executed, CPU starts by fetching the contents of location A in the main memory. Since indirect addressing is used, the value B stored in A is not the operand, but the address of the operand. Hence CPU requests another read operation from the main memory and this is to read the operands (contents of location B). The CPU then adds the operand to the contents of R0

In the second case, the operand is accessed indirectly through register R1 which contains the value B.

Note: The register or memory location that contains the address of the operand is called a pointer. Indirection is a powerful concept in programming.

5. Index mode: In this mode, the effective address of the operand is generated by adding a constant value to the contents of a register. The register used may be a special register provided for this purpose or may be any one of the general purpose register – referred to as an ***Index Register.***

Index mode is indicated symbolically as X(R*i*), where X denotes a constant value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by EA or Aeff = X + [R*i*]

In assembly language program, the constant X may be given either as an explicit number or as a name representing a numerical value.

There are two ways of using the index mode.

1. Offset is given as a constant: Here the index register R1 contains the address of a memory location and the value X defines an offset called displacement

   Add 20(R1),R2

2. Offset is in the index register: Here the constant X corresponds to a memory address and the contents of the index register define the offset to the operand.

   Add 1000(R1),R2

In either case, the effective address is the sum of two values; one is given explicitly in the instruction and the other is in a register.

5. Relative mode: The effective address is determined by the index mode. But here the program counter (PC) is used in place of the general purpose register R*i*. i.e. X(PC) can be used to address a memory location that is X bytes away from the location presently pointed by the program counter. Since the addressed location is identified "relative" to the program counter, which always identifies the current execution point in a program, this mode is called as Relative mode.

This mode is used to access data operands. It is commonly used to specify the target address in branch instruction.

6. Autoincrement mode: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of the register is automatically incremented to point to the next item in a list. We denote the autoincrement mode by putting the specified register in parenthesis to show that the contents of register is used as the effective address, followed by a plus (+) sign to indicate that these contents are to be incremented after the operand is accessed. Thus the autoincrement mode is written as

(R*i*) +. If we use autoincrement mode, it is possible to eliminate the increment instruction

7. Autodecrement mode: The contents of a register specified in the instruction are decremented. These contents are then used as the effective address of the operand. We denote the

autodecrement mode by putting the specified register in the parenthesis, preceded by a minus (-) sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus we write − (R4). This mode allows accessing of operands in the direction of descending address.

4.3 ASSEMBLY LANGUAGES

Assembler:

Programs written in assembly level language can be automatically translated into a sequence of machine instructions by a program called an assembler. Assembler is an

important utility program, and it is an example for system software. Like any other program, an assembler is also stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into a computer with the help of a keyboard and it is either stored in the memory of the computer or on the magnetic disk. So, a user program is simply a set of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it and then generates the desired machine language program. The user program in its original alphanumeric text is called as a source program, and the assembled machine language program is called as an object program.

Note: the assembly language for a given computer may or may not be case sensitive i.e. it may or may not distinguish between capital and lower case letters.

In our discussions, we will use capital letters to denote all symbolic names and labels. And the instruction format consists of the following:

Op-code Operand1, Operand2

The op-code (or operation code**)** mnemonic is followed by at least one blank space character and this is followed by the information that specifies the operands separated by comma. Since there are several addressing modes for specifying operand locations, the instruction must indicate this.

Examples:

1. MOVE R0, SUM

The mnemonic MOVE represents the binary pattern or OP code. This operation move is performed by this instruction. The assembler translates this mnemonic into binary OP code that is computer is capable of understanding. Here R0 is the source operand and SUM is the binary address representation of the destination operand and it is in the memory location.

2. ADD #5, R3

This instruction adds he number 5 to the contents of register R3. The result of this operation is put back in the register R3. # symbol indicates that the addressing mode

followed is immediate mode. Where in, the data value 5 is explicitly specified as a part of instruction. Some assembly language use op-code mnemonic instead of symbols. In such case, the instruction ADDI 5, R3 is used in place of ADD #5, R3 where the suffix I in the mnemonic ADDI states that the source operand is given in the immediate addressing mode.

3. MOVE #5, (R2)

In this instruction, the parentheses around the name or symbol denote a pointer to the operand. Suppose R2 contains MEM1 the address of memory location, then executing this instruction means the number 5 is to be placed in a memory location which is pointed to by register R2. This instruction can also be written as MOVEI 5(R2) where the suffix I in the mnemonic MOVEI states I denotes Indirect mode.

4.4 ASSEMBLER DIRECTIVES

Apart from providing a mechanism for representing instructions in a program, the assembly programming language also allows the programmer to specify other information which is essential to translate the source program into object program. Such statements in a program are called assembler directives or commands. This statement does not denote an instruction that will be executed when the object program is run. As an example let us consider the statement

SUM EQU 200

In the above statement, SUM is a name which is used to represent value 200. This statement informs the assembler that the name SUM should be replaced with the value 200 when ever it is called for.

Let us illustrate the use of assembly language through the following program. Here we are considering a 32 bit word length computer and it is byte addressable. The memory arrangement for the program is shown in Figure 4.1.

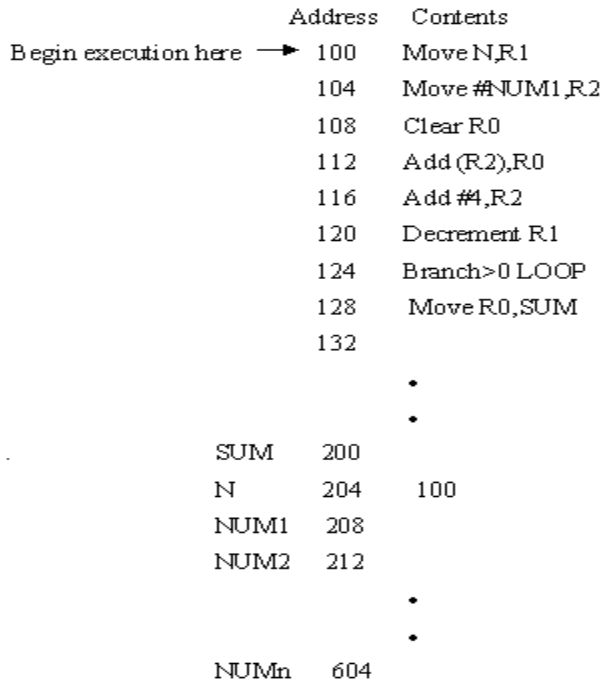


Figure 4.1 Memory arrangements for the program

Figure 4.1 shows the memory addresses where the machine instructions and the required data items are to be found after the program is loaded for the execution. The assembler needs to know the information pertaining to questions such as: how to interpret names, where to place the instructions in the memory or where to place the data operands in the memory, if it has to produce an object program for the program. The source program may be written as shown in Figure 4.2 to provide the required information for the assembler with respect to program in Figure 4.1. The use of assembler directives may be seen in example program of Figure 4.2.

	Memory address label	Operation	Addressing or data Information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directive		RETURN	START
		END	

Figure 4.2 assembly language representations for the program in Figure 4.1

The meaning and importance of the assembler directives used in the example program are given below:

1. EQU

EQU is the Equate directive. It informs the assembler about the value of SUM.

2. ORIGIN

This informs the assembler program where in the memory to place the data blocks that follows.

3. DATAWORD

This directive is used to inform the assembler about the data value that needs to be placed in the memory location indicated by ORIGIN directive.

4. RESERVE

This indicates that the specified size of memory block is to be reserved for data. In the example, a memory block of 400 bytes is to be reserved for data and the name NUM1 is to be associated with address 208.

5. END

This tells the assembler that this is the end of the source program text. It includes the label START which is the address of the location in which execution of the program is to begin.

6. RETURN

This directive identifies the point at which execution of the program should be terminated. It causes the assembler to insert an appropriate machine instruction that returns control to the operating system of the computer.

4.5 ASSEMBLY AND EXECUTION OF PROGRAMS

A source program which is written in an assembly language must be assembled into machine language object program before it gets executed. This task is performed by an assembler which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions and replaces all names and labels with their actual values. The assembler assigns addresses to instructions and data blocks starting at the address given in the ORIGIN assembler directives. It also inserts constants that may be given in the DATAWORD directives and it reserves memory space as requested by the RESERVE commands.

An important part of the assembly process is to determine the values that replace the names. The value of a name can be specified by an EQU directive or a name can be defined in the Label field of a given instruction. The value represented by the name is determined by the location of the instruction which is currently under consideration in the assembled object program. Hence, the assembler must be able to keep track of addresses as it generates the machine code for successive instructions.

In certain instances, for example when there is a branch instruction in the program, the assembler do not directly replace a name representing an address with the actual value of this address. Instead, it implements the branch instruction by specifying the branch target using relative addressing mode. The assembler computes the branch offset, which is nothing nut the distance to the target and puts it into the machine instruction.

Two-pass assembler:

An assembler keeps track of all names and numerical values that correspond to them while scanning through a source program with the help of a symbol table. When a name appears for the second time, it is replaced with its value from the symbol table. But, a problem arises when a name appears as an operand before it is given a value. A simple solution to this problem is to have the assembler scan through the source program two times. During the first scan or first pass, it creates a complete symbol table. At the end of this pass, all names are assigned with numerical values. The assembler then goes through the source program for a second time or second pass where in it substitutes values for all names from the symbol table. This kind of assembler with two passes is known as a two-pass assembler.

Loader:

The assembler stores the object program on a magnetic disk. The object program must be loaded into the memory of a computer before it is being executed. In order to accommodate this, a loader is used. A loader is a utility program placed in memory. Executing the loader performs a sequence of input operations that are needed to transfer the machine language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will

be stored. This information is made available by an assembler and it is placed as information in a header preceding the object code. When the object program begins executing, it proceeds to completion unless there are logical errors in the program. And these errors are found out by the users with the help of system software program known as a debugger. This program enables a user to stop execution of the object program at certain points of interest and also to examine the contents of various processor registers and memory locations.

4.6 STACKS AND QUEUES

A stack is an important data structure. It is a list of data items with the accessing restriction that item can be added or removed at one end of the list only. This is called the top of the stack, and the other end is called the bottom. A stack is also called pushdown stack or LIFO data structure. E.g, a pile of trays in cafeteria, the way they are used and placed back. The terms push and pop are used to describe placing a new item on the stack and removing the item from the stack, respectively.

Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

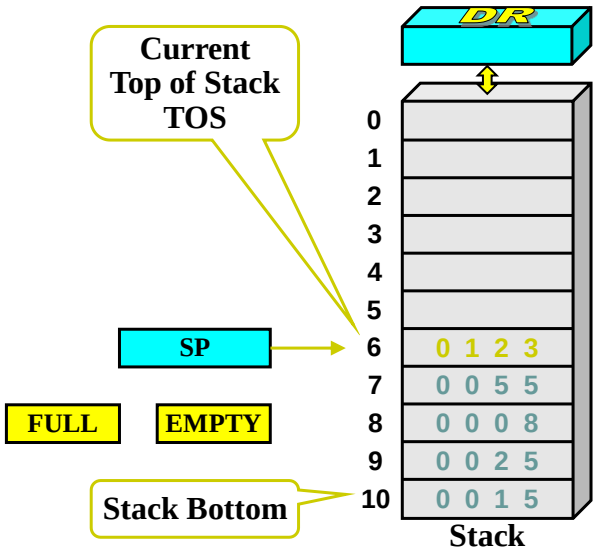


Figure 4.3 A stack of words in the memory.

Figure 4.3 shows a stack of word data items in the memory of a computer. It contains numerical values, with 43 at the bottom and −28 at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the *stack pointer* (SP). It could be one of the general-purpose registers or a register dedicated to this function. The push operation can be implemented as

```
Subtract #4, SP
Move NEWITEM,(SP)
```

where the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move. The pop operation can be implemented as

```
Move (SP), ITEM
Add #4, SP
```

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element. If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be performed by the single instruction

Move NEWITEM,−(SP)

`

and the pop operation can be performed by

Move (SP)+,ITEM

When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we must avoid pushing an item onto the stack when the stack has reached its maximum size. Also, we must avoid attempting to pop an item off an empty stack, which could result from a programming error

Another useful data structure that is similar to the stack is called a *queue*. Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So, two pointers are needed to keep track of the two ends of the queue.

Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a *circular buffer*. Let us assume that memory addresses from BEGINNING to END are assigned to

the queue. The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence, the back pointer is reset to the value BEGINNING and the process continues.

As in the case of a stack, care must be taken to detect when the region assigned to the data structure is either completely full or completely.

4.7 NUMBER REPRESENTATION AND OPERATIONS

While programming using an assembly language it is convenient to use any familiar number representations for representing numerical values. These values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most of the assemblers allow the programmer to specify numeric values in various ways, using conventions that are defined by the syntax of assembly language.

Let us consider an example, the number 93 which can be represented by an 8-bit binary number 01011101. If this value is to be used as an immediate operand, it can be written as a decimal number as shown in the instruction

```
ADD #93,R1
```

or as a binary number identified by a prefix symbol such as with a percent sign. It is shown as

```
ADD #%01011101, R1
```

But writing instructions using binary numbers become very cumbersome. So, to make the program development easy and convenient, the binary numbers are written in a compact manner by using hexadecimal number representations. In hexadecimal or hex notation, four bits are represented by a single hex digit. This notation is a direct extension of the Binary Coded Decimal (BCD) code. Where, the numbers 0000, 0001, ……,1001 are represented by the digits 0, 1, …..,9 as in BCD. The remaining six 4-bit patterns 1010, 1011,……,1111 are represented by letters A, B, …., F. In hexadecimal representation, the

decimal value 93 becomes 5D and it is often identified by prefixing a dollar sign. Thus the instruction ADD #%01011101, R1 could be written in assembly language with hex notation as

ADD #$5D,R1

Check your progress:

1. Explain the following addressing modes using suitable examples: absolute mode, index mode, immediate mode, relative mode
2. What is an assembler?
3. With an example, explain the use of assembler directives.
4. What is hexadecimal notation? Give examples

4.8 SUMMARY

In this unit, the principles of general addressing techniques were emphasized. We were able to see and briefly discuss about the most important system software such as an assembler and a loader. An assembler translates the assembly language program into machine language. Using mnemonics it is possible to write assembly language programs. Further, assembler directives or commands are used by the programmers for specifying other information to an assembler for translating the source program into object program. The manner in which an assembly language program is written and executed is briefly discussed in this unit along with the importance of representing numerical values in hex notation.

4.9 KEY WORDS

Addressing mode: The method used to provide an access path to operands in memory and CPU registers.

Effective address: The address generated by the CPU to access the operands in memory

Mnemonics: Symbolic names or acronyms used for representing patterns of 0's and 1's in an instruction.

Assembly Language: Is a programming language with a complete set of mnemonics and the rules for using it.

Assembler: Is a program which translates the program written in assembly language into a sequence of machine instructions.

Source program: The user program in its original alphanumeric text format.

Object program: The assembled machine language program.

Assembler directives: Directives or commands used by the assembler while it translates a source program into an object program.

4.10 ANSWERS TO CHECK YOUR PROGRESS

1. 4.2
2. 4.3
3. 4.4
4. 4.7

4.11 UNIT-END EXERCISES AND ANSWERS

1. What is autoincrement mode? When do you use it?
2. What are mnemonics? Give examples.
3. Define the following: Source program, object program, assembler
4. Explain the following assembler directives: ORIGIN, DATAWORD, RESERVE
5. What is a two-pass assembler? Explain the importance of each phase.
6. What is the function of: loader, debugger
7. Expand BCD.

Answers: SEE

1. 4.2
2. 4.1
3. 4. 3
4. 4.4

5. 4.5
6. 4.5
7. 4.7

4.12 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002.
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006.
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 5: FUNDAMENTAL CONCEPTS

Structure

5.0 OBJECTIVES

At the end of this unit you will be able to

- Understand some Fundamental concepts such as
- Register transfers
- Performing an Arithmetic or Logic Operation
- Fetching a word from memory and storing a word in memory
- Understand execution of a complete instruction
- Explain branch instruction

5.1 INTRODUCTION

This unit is about the processing unit, which executes machine instructions and coordinates the activities of other units. This is also called a processor or instruction set processor (ISP). We understand its internal structure and how it performs the tasks of fetching, decoding, and executing instructions of a program. The processing unit is called central processing unit (CPU). We explore the organization of the hardware that enables a CPU to perform its main function. We learn how the execution of a complete instruction takes place and we also learn Branch Instructions.

5.2 SOME FUNDAMENTAL CONCEPTS

A program, a set of instructions, to be executed by a computer is loaded in sequential locations in the main memory. To execute this program, the CPU fetches one instruction at a time and performs the functions specified. Until a branch or a jump instruction is executed, instructions are fetched from successive memory locations. The address of the next instruction to be executed is kept by the CPU in a dedicated register called program counter (PC). The contents of the PC are updated to point to the next instruction in the sequence.

Assume that each instruction occupies one memory word. Therefore, one instruction execution requires the CPU to perform the following 3 steps:

1. Fetch the contents of the memory location by the PC into instruction register (IR). Symbolically, this can be written as:

   $$IR \leftarrow [[PC]]$$

2. Increment the contents of the PC by 1, i.e., (assuming word addressable)

PC ← [PC]+1

3. Carry out the actions specified by the instruction in the IR.

Steps 1 and 2 are called the fetch phase and step 3 is called the execution phase.

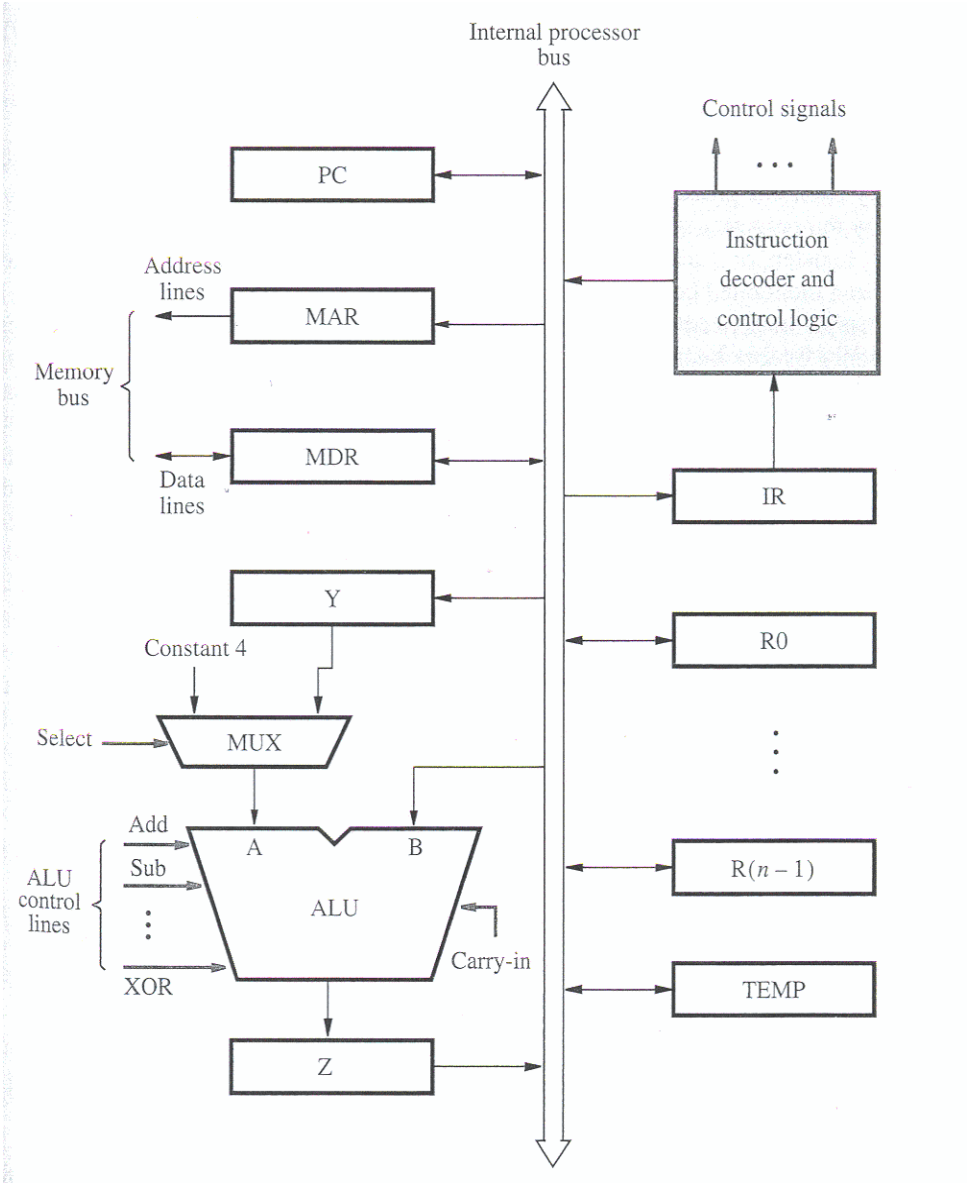


Figure 5.1 Single-bus Organization of the Datapath inside a processor

Figure 5.1 shows an organization in which the arithmetic and logic unit (ALU) and all the registers are interconnected via a single common bus. This bus is internal to the processor.

The address and data lines of the external memory bus are shown in Figure 5.1 connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

The use and number of the processor registers R0 through R(n-1) vary considerably from one processor to another. Registers may be provided for general purpose use by the programmer. Some may be dedicated as special-purpose registers, such as index registers or stack pointers. Three registers, Y, S, and TEMP in Figure 5.1 are transparent to the programmer. i.e., the programmer need not be concerned with them, because they are never referenced explicitly by any instruction. They are used by the processor for temporary storage during execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction.

The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. We will refer to the two possible values of the MUX control input Select as Select4 and Select Y for selecting the constant 4 or register Y, respectively.

As instruction execution progresses, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation. The instruction decoder and control logic unit is responsible for implementing the actions specified by the instruction loaded in the 1R register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data. The registers, the ALU, and the interconnecting bus are collectively referred to as the *datapath*.

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

1. Fetch the contents of a given memory location and load them into a processor register.
2. Store a word of data from a processor register into a given memory location.
3. Transfer a word of data from one processor register to another or to the ALU.
4. Perform an arithmetic or logic operation and store the result in a processor register.

5.2.1 Fetching a Word from memory

To fetch a word from memory, the CPU has to specify the address of the memory location where this information is stored and request a read operation. The CPU transfers the address of the required word of information to the MAR, which is connected to address lines of the memory bus. The CPU uses the control lines of the memory bus to indicate a Read operation is needed. Then the CPU waits for Read operation completion, which is indicated by Memory-Function Completed (MFC) signal set. When the MFC is set, the information on the data lines is loaded into MDR.

The connections for register MDR are illustrated in Figure 2.4. It has four control signals: MDR_{in} and MDR_{out} control the connection to the internal bus, and $MDR_{in\ E}$ and $MDR_{out\ E}$ control the connection to the external bus.

The example below demonstrates how to fetch a word from memory location, whose address is specified in R1, and place the word fetched in R2

1 MAR ← [R1]
2 Request memory READ and put the data to the address register
3 Wait for the Memory Fetch Cycle (MFC) signal and put the result from [MDR] to R2.
4 R2 ←[MDR]

Both 2 and 3 are regarded as asynchronous data transfer.

A data transfer in which one device initiates the transfer and waits until the other device responds (with an MFC signal) is referred to as an asynchronous transfer. An alternative scheme in many computers is synchronous. In synchronous transfer, one of the control lines of the bus carries pulses from a clock running continuously at a fixed frequency. These pulses provide common timing signals to the CPU and the main memory.

5.2.2 Storing a Word in Memory

After the address is loaded into MAR and data into MDR, The CPU uses the control lines of the memory bus to indicate a Write operation is needed.

The example below shows how the machine store a word in R2 into a memory location, whose address is specified in R1

1 MAR ← [R1]
2 MDR ←[R2]
3 Request memory write
4 Wait for MFC signal

Both steps 3 and 4 are regarded as asynchronous data transfer. As in the case of the Read operation, the Write control signal causes the memory bus interface hardware to issue a

Write command on the memory bus. The processor remains in step 3 until the memory operation is completed and an MFC response is received.

5.2.3 Register Transfers

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register. This is represented symbolically in Figure 5.2 the input and output of register R_i are connected to the bus via switches controlled by the signals Ri_{inn} and Ri_{out} respectively. When Ri_{inn} is set to 1, the data on the bus are loaded into R_i. Similarly, when Ri_{out} is set to 1, the contents of register R_i are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows:

- Enable the output of register R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the processor bus.
- Enable the input of register R4 by setting $R4_{in}$ to 1. This loads data from the processor bus into register R4.

All operations and data transfers within the processor take place within time periods defined by the processor clock. The control signals that govern a particular transfer are asserted at the start of the clock cycle. In our example, $R1_{out}$ and $R4_{in}$ are set to 1. The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock, the fip-flops that constitute R4 will load the data present at their inputs. At the same time, the controls signals $R1_{out}$ and $R4_{in}$ will return to 0.

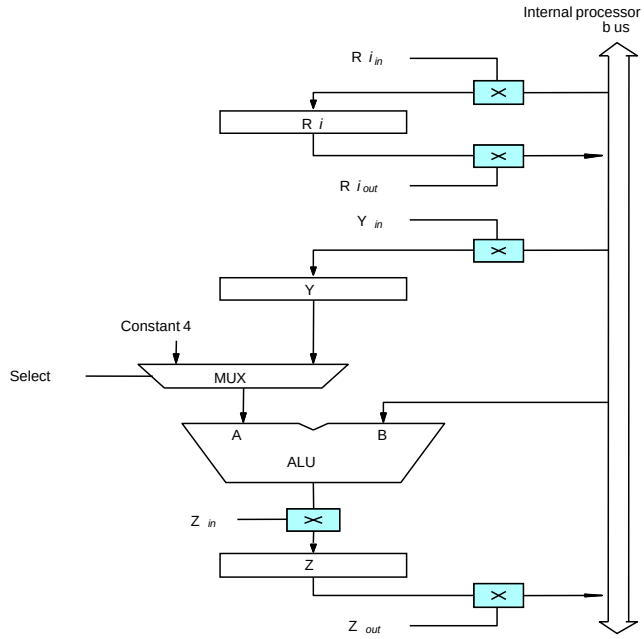


Figure 5.2 Input and output gating for the registers in Figure 5.1.

5.2.4 Performing an Arithmetic or Logic Operation

The ALU is a combinational circuit that has no internal storage. It performs arithmetic and logic operations on the two operands applied to its' A and B inputs. To add two numbers, the two operands have to be made available at the inputs of the ALU simultaneously. In Figures 5.1 and 5.2, one of the operands is the output of the multiplexer MUX and the other operand is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z. Therefore, a sequence of operations to add the contents of register R1 to those of register R2 and store the result in register R3 is:

1. $R1_{out}$, Y_{in}
2. $R2_{out}$, Select Y, Add, Z_{in}
3. Z_{out}, $R3_{in}$

The signals whose names are given in any step are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive. Hence, in step 1, the output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y. In step 2, the multiplexer's Select signal is set to Select Y, causing the multiplexer to gate the contents of register Y to input A of the ALU. At the same time, the contents of register R2 are gated onto the bus and, hence, to input B. The function performed by the ALU depends on the signals applied to its control lines. In this case, the Add line is set to 1, causing the output of the ALU to be the sum of the two numbers at inputs A and B. This sum is loaded into register Z, because its input control signal is activated. In step 3, the contents of register Z are transferred to the destination register, R3. This last transfer cannot be earned out during step 2, because only one register output can be connected to the bus during any clock cycle.

5.2.1 Register Gating and Timing of Data Transfers

`Let us consider the case of each bit of the registers in Figure 5.1 and 5.2 consists of a flip-flop as shown in Figure 5.3. The flip-flop shown is assumed to be one of the bits of register Z. When the control input Z_{in} is equal to 1, the flip-flop state changes to corresponding to the data on the bus. Following a 1 to 0 transition at the Z_{in} input, the data stored in the flip-flop immediately before this transition is locked in until Z_{in} is again set 1.

Figure 5.3 Input and output gating for one register bit

5.3 EXECUTION OF A COMPLETE INSTRUCTION

Let us now consider the sequence of elementary operations required to execute one instruction. Consider the instruction,

Add (R3), R1

which adds the contents of a memory location pointed to by R3 to register R1. Executing this instruction requires the following actions:

1. Fetch the instruction
2. Fetch the first operand (the contents of the memory location pointed to by R3)
3. Perform the addition
4. Load the result into R1

The Figure 5.4 gives the sequence of control steps required to perform these operations for the single-bus architecture of **Figure 5.1**. Instruction execution is as follows: In step 1, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select1, which causes the multiplexer MUX to select the constant 1. This value is added to the operand at input B, which is the contents of the PC and the result is stored in register. The updated value is moved from register back into the PC during step 2, while waiting for the memory to respond, the word fetched from the memory is loaded into the IR.

Step	Action
1.	PC_{out} , MAR_{in} , Read, Select1, Add, Z_{in}
2.	Z_{out} , PC_{in} , Y_{in} , WMFC
3.	MDR_{out} , IR_{in}
4.	$R3_{out}$, MAR_{in} , Read
5.	$R1_{out}$, Y_{in} , WMFC
6.	MDR_{out} , SelectY, Add Z_{in}
7.	Z_{out} , $R1_{in}$, End

Figure 5.4 Control Sequence for Execution of the instruction ADD (R3), R1

Steps 1 through 3 constitute the instruction fetch phase, which is the same for all instructions. The instruction decoding circuit interprets the contents of the IR at the beginning of step 4. This enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase. The contents of register R3 are transferred to the MAR in step 4 and a memory Read operation is initiated. Then, the contents of R1 are transferred to register Y in step 5, to prepare for the addition operation. When the Read operation is completed, the memory operand is available in register MDR and the addition operation is performed in step 6. The contents of MDR are gated to the bus and thus also to the B input of the ALU and register Y is selected as the second input to the ALU by choosing Select Y. The sum is stored in register Z and then transferred to R1 in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

5.3.1 Branch Instructions

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X which is given in the branch instruction to the updated value of the PC. Figure 5.5 gives a control sequence that implements an unconditional branch instruction.

Step	Action
1.	PC_{out} , MAR_{in} , Read, Select1, Add Z_{in}
2.	Z_{out} , PC_{in} , Y_{in} , WMFC
3.	MDR_{out} , IR_{in}
4.	Offset-field-of-IR_{out} , Add Z_{in}
5.	Z_{out} , PC_{in} , End

Figure 5.5 Control Sequence for an unconditional Branch Instruction

Processing starts, as usual with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3. The offset value is extracted from the IR by the instruction

decoding circuit which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4 and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

The offset X used in branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction. For example, if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 49. The reason for this can be readily appreciated from the control sequence in the Figure 5.5. The PC is incremented during the fetch phase, before knowing the type of instruction being executed. Thus, when the branch address is computed in step 4, the PC value used is the updated value, which points to the instruction following the branch instruction in the memory.

Consider now a conditional branch. In this case, we need to check the status of the condition codes before loading a new value into the PC. For example, for a Branch-on-negative (Branch<0) instruction, step 4 in Figure 5.5 is replaced with:

Offset-field-of-IR_{out}, Add, Z_{in} , If N=0 then End

Thus, if N=0, the processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

Check your progress

1 With a neat diagram of single bus organization, explain the working of PC.

2 What is register transfer?

3 Give control Sequence for Execution of the instruction ADD (R3), R1.

5.4 SUMMARY

In this unit we have learnt some fundamental Concepts such as Register Transfers, Performing an Arithmetic or Logic Operation and Fetching a word from Memory and Storing a word to memory. We also learnt how the execution of a complete instruction and also branch instruction is carried out in the computing system.

5.5 KEYWORDS

MAR: Memory Address Register,

MDR: Memory Data Register

PC – Program Counter

IR – Instruction Register

MDR – Memory Data Register

MAR – Memory Address register

5.6 ANSWERS TO CHECK YOUR PROGRESS

1 5.2

2 5.2.3

3 5. 3

5.7 UNIT END EXERCISES AND ANSWERS

1. Elucidate Fundamental Concepts.
2. Elucidate Register Transfers,
3. Sketch out performing of Arithmetic and Logic Operation
4. Explain fetching a word from Memory.
5. Discuss about execution of a complete instruction.
6. Explain branch instruction.

Answer: SEE

1 5.2

2 5.2.3

3 5.2.4

4 5.2.1

5 5.3

6 5.3.1

5.8 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 6: PERFORMANCE CONSIDERATIONS

Structure

6.0 OBJECTIVES

After studying this unit, we will be able to:

- Explain Multiple Bus Organization
- Understand need for overlapping fetch and execution operations
- Realize the usage of Cache

6.1 INTRODUCTION

In this unit, we will learn, Multiple Bus Organization and also the control sequence for the instruction. This unit also explains the other performance enhancements possible.

We have used the simple single-bus structure in Figure 5.1 to elucidate the basic ideas. The resulting control sequences in Figures 5.6 and 5.7 are quite long because only one data item can be transferred over the bus in a clock cycle. To reduce the number of

steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel.

Performance of a computer depends on many factors, some of which are related to the design of the CPU. Three of the most important factors are the power of the instruction, the clock cycle time, and the number of clock cycle per instruction.

A powerful instruction performs a complex multistep task, at the cost of several clock cycles for execution. The question is to have complex instructions or simple instruction. But, the evolution of RISC processors has demonstrated that it may be advantageous to use simple instructions.

Clock speed has a major influence on performance. It depends on the technology used to implement the electronic circuits and the complexity of functional units such as the ALU.

So far we considered a simple single bus model. For better performance, we now consider more complex structures. It is always desirable to use as few clock cycles as possible. A single clock cycle per instruction is ideal. This cannot be achieved with simple model, because bus allows only one data item to be transferred during one clock cycle. Therefore, we should consider the use of multiple buses within the CPU.

6.2 MULTIPLE BUS ORGANIZATION

Figure 6.1 depicts a three-bus structure used to connect the registers and the ALU of a processor. All general-purpose registers are combined into a single block called the *register file*. In VLSI technology, the most efficient way to implement a number of registers is in the form of an array of memory cells similar to those used in the implementation of random-access memories (RAMs). The register file in Figure 6.1 is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B.

The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Figure 6.1: Three-bus organization of the datapath

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z in Figures 5.1.

A second feature in Figure 6.1 is the introduction of the Incrementer unit, which is used to increment the PC by 1. Using the Incrementer, eliminates the need to add 1 to the PC using main ALU, as was done in Figures 5.6 and 5.7. The source for the constant 1 at the ALU input multiplexer is still useful. It can be used to increment other addresses such as the memory addresses in Load Multiple and Store Multiple instructions.

The structure in Figure 6.1 requires significantly fewer control steps to execute instructions compared to Figure 5.1. Consider the three-operand instruction of the form

OP Rsrc1, Rsrc2, Rdst

in which an operation is performed on the contents of two source registers, and the result is placed into a destination register. Buses A and B are used to transfer the source operands, and bus C provides the path to the destination. The path from the source buses to the destination bus goes through the ALU, where the required operation is performed. Thus, assuming that the operation to be performed can be completed in one pass through the ALU, the structure of Figure 6.1 allows the execution phase of an instruction to be performed in one cycle. Note that if it is merely necessary to copy the contents of one register into another, then the transfer is also done through the ALU, but no arithmetic or logic operation is performed.

The temporary storage registers Y and Z in Figure 5.1 are not required in Figure 6.1. Register Y is not needed because both inputs to the ALU are provided simultaneously via buses A and B. Register Z is not needed because the output from the ALU is transferred to the destination register via the third bus, C. In this structure it is essential to ensure that the same register can serve as both the source and the destination in a given instruction. This would not be possible if the registers were simple latches as in Figure 5.3. Instead the register file must be implemented using either edge-trigger1ed or master–slave circuits.

Example 6.1: Consider

Add R4, R5, R6

The control sequence for executing this instruction is given in the Figure 6.2. In step 1, the contents of the PC are passed through the ALU using the R=B control signal and loaded into the MAR to start a memory read operation. At the same time, the PC is incremented by 1. Note that the value loaded into MAR is the original contents of the PC.

The incremented value is loaded into the PC at the end of the clock cycle and will not affect the contents of MAR. In step 2, the processor waits for MFC and loads the data received into MDR and then transfers them to IR in step 3. Finally, the execution phase of the instruction requires only one control step to complete step 4.

Step	Action
1.	PC_{out} , R=B, MAR_{in} , Read, IncPC
2.	WMFC
3.	MDR_{out} , R=B, IR_{in}
4.	$R4_{outA}$, $R5_{outBt}$, SelectA, Add, $R6_{in}$, End

Figure 6.2: Control Sequence for the instruction Add R4, R5, R6

By providing more paths for data transfer, a significant reduction in the number of clock cycles needed to execute an instruction is achieved.

The three-bus structure allows execution of register-to-register operation in a single clock cycle. This is particularly well suited to the requirements of RISC processors, in which most arithmetic and logic instructions have register operands.

6.3 OTHER ENHANCEMENTS

It is possible to improve Performance greatly, if the CPU can overlap the fetch and execute phases of instructions. While one instruction is being executed, the next instruction can be pre-fetched from the memory. Recent processors include special instruction unit, which fetches instructions and places them into a queue ready for execution. The instruction unit generates memory addresses based on the address of the last instruction fetched. It attempts to ensure that correct instructions are pre-fetched when a branch instruction is encountered.

Use of a WMFC signal to wait for the response from a main memory is slower than the CPU. Another approach to improve the performance is by use of cache memory on the

same chip as the CPU. Data can be accessed from the cache in one clock cycle. Hence, if the required instructions and data are usually found in the cache, the apparent memory access time will be short. If the desired data are not found in the cache (cache miss), it is necessary to access the data in the main memory, which takes more time.

6.4 A COMPLETE PROCESSOR

A complete processor can be designed using the structure shown in Figure 6.3. This structure has an instruction unit that fetches instructions from an instruction cache or from the main memory when the desired instructions are not already in the cache. It has separate processing units to deal with integer data and floating-point data. Each of these units can be organized as shown in Figure 6.1.

Figure 6.3: Block diagram of a complete Processor

A data cache is inserted between these units and the main memory. Using separate caches for instructions and data is common practice in many processors today. Other processors use a single cache that stores both instructions and data. The processor is

connected to the system bus and hence, to the rest of the computer, by means of a bus interface.

Although, we have shown just one integer and one floating-point in Figure 6.3, a processor may include several units of each type to increase the potential for concurrent operations.

Check your progress

1. Compare single bus CPU with multiple CPU
2. Why to overlap fetch and execution operations
3. What is the use of cache memory?

6.6 SUMMARY

In this unit, we have learnt the working of Multiple Bus Organization. We also discussed about other enhancement to improve performance. We ended the unit with a block diagram of a complete CPU.

6.5 KEYWORDS

MUX – Multiplexer

IR – Instruction Register

MDR – Memory Data Register

ALU – Arithmetic and Logic Unit

6.6 ANSWER TO CHECK YOUR PROGRESS

1. 6.1
2. 6.3
3. 6.3

6.7 UNIT_END EXCERCISES AND ANSWERS

1. Elucidate Multiple Bus Organization, with a neat diagram.
2. What are the other enhancements to improve performance of a computer
3. With block diagram, explain a complete CPU.

Answer: See:

1. 6.2
2. 6.3
3. 6.4

6.8 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 7: HARD-WIRED CONTROL

Structure

7.0 OBJECTIVES

After studying this unit, we will be able to

- Explain Hard-Wired Control.

7.1 INTRODUCTION

In this unit, we learn how Hardware is used for generating internal control signals. To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. A variety of techniques have been used to organize a control unit. Most of them fall into two major categories:

1. Hardwired control organization
2. Microprogrammed control organization

In this unit we discuss hardwired control. In the hardwired organization, the control unit is designed as a combinational circuit. That is, the control unit is implemented by gates, flip-flops, decoder and other digital circuits. Hardwired control units can be optimized for fast operations

7.2 HARD – WIRED CONTROL

Consider the sequence of control signals in Figure 5.4. Each step in this sequence is completed in one clock period. A counter may be used to keep track of the control steps, as shown in Figure 7.1. Each state, or count, of this counter corresponds to one control step. The required control signals are determined by the following information:

- Contents of the control step
- Contents of the instruction register
- Contents of the condition code flags
- Other status flags (such as MFC)

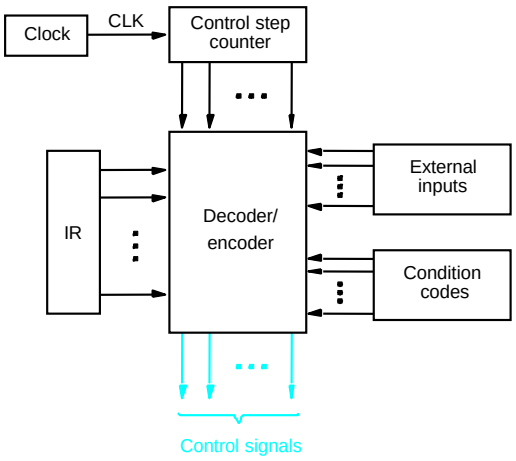


Figure 7.1 Control unit organization

To understand the structure of the control unit, we start with a simplified view of the hardware involved. The decoder/encoder block in the above Figure 7.1 is a combinational circuit that generates the required control outputs, depending on the state of all its inputs. By separating the decoding and encoding functions, we obtain the more detailed block diagram in Figure 7.2. The step decoder provides a separate signal line for

each step, or time slot, in the control sequence. Similarly, the output of the instruction decoder consists of a separate line for each machine instruction. For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1 and all other lines are set to 0. The input signals to the encoder block in Figure 7.2 are combined to generate the individual control signals Y_{in} PC_{out} Add, End and so on. An example of how the encoder generates the Z_{in} control signal for the processor organization in Figure 5.1 is given in Figure 7.3. This circuit implements the logic function

$$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + ...$$

Figure 7.2 Generation of the decoding and encoding functions

This signal is asserted during time slot T_1 for all instructions, during T_6 for an Add instruction, during T_4 for an unconditional branch instruction, and so on. The logic function for Z_{in} is derived from the control sequences in Figures 5.4 and 5.5. As another example, Figure 7.4 gives a circuit that generates the End control signal from the logic function

$$End = T_1 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{.N}) \cdot BRN + \ldots.$$

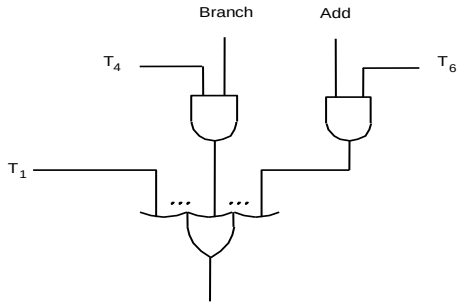


Figure 7.3 Generation of the Z_{in} Control signal

Figure 7.4 Generation of the End Control signal

The End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value. Figure 7.2 contains another control signal called RUN. When set to 1, RUN causes the counter to be incremented by one at the end of every clock cycle. When RUN is equal to 0, the counter stops counting. This is needed whenever the WMFC signal is issued, to cause the processor to wait for the reply from the memory.

The control hardware shown in Figure 7.1 or 7.2 can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes and the external inputs. The outputs of the state machine are the control signals. The sequence of operations carried out by this machine is determined by the wiring of the logic elements, hence the name "hardwired". A controller that uses this approach can operate at high speed. However, it has little flexibility and the complexity of the instruction set it can implement is limited.

Check your progress

4. What are the major categories of organizing control unit?
5. With diagram, explain Control unit organization.
6. Explain generation of the End Control signal.

7.6 SUMMARY

In this unit, we have learnt Hard-wired control.

7.5 KEYWORDS

Hardwired: The sequence of operations carried out is determined by the wiring of the logic elements.

7.6 ANSWER TO CHECK YOUR PROGRESS

4. 7.1

5. 7.2
6. 7.2

7.7 UNIT_END EXCERCISES AND ANSWERS

4. Elucidate generation of the decoding and encoding functions.

2 Explain the generation of the Z_{in} Control signal.

Answer: See

1. 7.2
2. 7.2

7.8 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 8: MICROPROGRAMMED CONTROL

Structure

8.0 OBJECTIVES

After studying this unit, we will be able to explain:

- Microprogrammed Control
- Microinstructions
- Microprogramming Sequencing
- Pre fetching Microinstructions
- Emulation

8.1 INTRODUCTION

In this unit, we will learn Microprogramming approach and Microprogram organization. To carry out the execution of instructions, the processor must have some means of

generating the control signals needed in the proper sequence. Designers employ a wide variety of techniques to this purpose. There are two categories of approaches: hardwired control and microprorgam control. In the rest of this unit, we will focus on Microprogrammed approach.

8.2 BASICS OF MICROPROGRAMMED CONTROL

We explain here an alternative scheme called *micro programmed control*, in which control signals are generated by a program similar to machine language programs.

Figure 8.1 An example of microinstruction

We start with introducing some common terms. A *control word* (CW) is a word whose individual bits represent the various control signals in Figure 7.2. Each of the control steps in the control sequence of an instruction defines a unique combination of 1s and 0s in the CW. The CWs corresponding to the 7 steps of Figure 5.4 are shown in Figure 8.1. We have assumed that Select Y is represented by Select = 0 and Select1 by Select = 1. A sequence of CWs corresponding to the control sequence instruction

constitutes the *micro-routine* for that instruction, and the individual control words in this micro-routine are referred to as *microinstructions.*

The micro-routines for all instructions in the instruction set of a computer are stored in a special memory called the *control store*. The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding micro-routine from the control store. This suggests organizing the control unit as shown in Figure 8.2. To read the control words sequentially from the control store, a *microprogram counter* (μPC) is used. Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the μPC. The μPC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store. Hence, the control signals are delivered to various parts of the processor in the correct sequence.

Figure 8.2 Basic organization of a microprogrammed control unit

There is one important function of the control unit that cannot be implemented by the simple organization in Figure 8.2. This is the situation that arises when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action. In the case of hardwired control, this situation is

handled by including an appropriate logic function, in the encoder circuitry. In micro programmed control, an alternative approach is to use conditional branch microinstructions. In addition to the branch address, these microinstructions specify which of the external inputs, condition codes, or possibly bits of the instruction register, should be checked as a condition for branching to take place.

The instruction Branch-on-negative (Branch<0) may now be implemented by a micro-routine such as that shown in Figure 8.3. After loading this instruction into IR, a branch microinstruction transfers control to the corresponding micro-routine, which is assumed to start at location 25 in the control store. This address is the output of the starting address generator block in Figure 8.2. The microinstruction at location 25 tests the N bit of the condition codes. If this bit is equal to 0, a branch takes place to location 0 to fetch a new machine instruction. Otherwise, the microinstruction at location 26 is executed to put the branch target address into register Z, as in step 4 in Figure 5.5. The microinstruction in location 27 loads this address into the PC.

Address	**Microinstruction**
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Zout, PCin, Yin, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 8.3 Microroutine for the microinstruction Branch<0

To support microprogram branching, the organization of the control unit should be modified as shown in Figure 8.4.

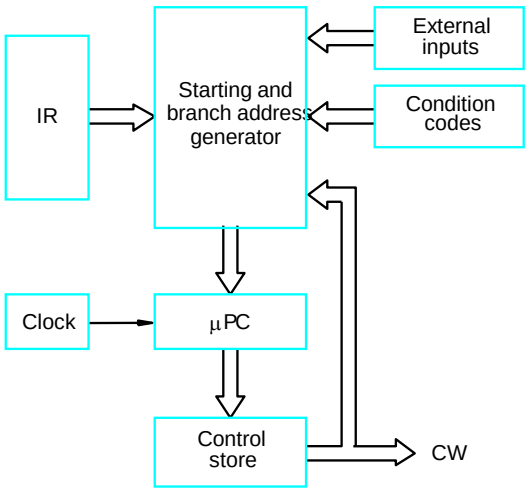


Figure 8.4 Organization of the control unit to allowconditional branching in the microprogram

The starting address generator block of Figure 8.2 becomes the starting and branch address generator. This block loads a new address into the μPC when a microinstruction instructs it to do so. To allow implementation of a condition branch, inputs to this block consist of the external inputs and condition codes as well as the contents of the instruction register. In this control unit, the μPC is incremented every time a new microinstruction is fetched from the microprogram memory, except in the following situations:

1. When a new instruction is loaded into the IR, the μPC is loaded with the starting address of the microroutine for that instruction,
2. When a Branch microinstruction is encountered and the branch condition is satisfied, the μ PC is loaded with the branch address.
3. When an End microinstruction is encountered, the μ PC is loaded with the address of the first CW in the microroutine for the instruction fetch cycle (this address is 0 in Figure 8.3)

8.3 MICROINSTRUCTIONS

After understanding a scheme for sequencing microinstruction, we now take a closer look at the format of individual microinstructions. A straightforward way to structure microinstructions is to assign one bit position to each control signal, as in Figure 8.1.

However, this scheme has one serious drawback – assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large. Moreover, only a few bits are set to 1 (to be used for active gating) in any given microinstruction, which means the available bit space is poorly used. Consider again the simple processor of Figure 5.1 and assume that it contains only four general purpose registers R0, R1, R2, and R3. Some of the connections in this processor are permanently enabled, such as the output of the IR to the decoding circuits and both inputs to the ALU. The remaining connections to various register require a total of 20 gating signals. Additional control signals not shown in the figure are also needed, including the Read, Write, Select, WMFC and End signals. Finally, we must specify the function to be performed by the ALU. Let us assume that 16 functions are provided, including Add, Subtract, AND, and XOR. These functions depend on the particular ALU used and do not necessarily have a one-to-one correspondence with the machine instruction OP codes. In total, 42 control signals are needed.

If we use the simple encoding scheme described earlier, 42 bits would be needed in each microinstruction. Fortunately, the length of the microinstructions can be reduced easily. Most signals are not needed simultaneously, and many signals are mutually exclusive. For example, only one function of the ALU can be activated at a time. The source for a data transfer must be unique because it is not possible to gate the contents of two different registers onto the bus at the same time. Read and Write signals to the memory cannot be active simultaneously. This suggests that signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one *microoperation* per group is specified in any instruction. Then it is possible to use a binary coding scheme to represent the signals within the group. For example, four bits suffice to represent the 16 available functions in the ALU. Register output control signals

can be placed in a group consisting of PC_{out}, MDR_{out}, Z_{out}, $Offset_{out}$, $R0_{out}$, $R1_{out}$, $R2_{out}$, $R3_{out}$ and $TEMP_{out}$. Any one of these can be selected by a unique 4-bit code.

Figure 8.5 An example of a partial format for field-encoded microinstruction

Further, natural groupings can be made for the remaining signals. Figure 8.5 shows an example of a partial format for the microinstructions in which each group occupies a field large enough to contain the required codes. Most fields must include one inactive code for the case in which no action is required. For example, the all-zero pattern in F1 indicates that none of the registers that may be specified in this field should have its content placed on the bus. An inactive code is not needed in all fields. For example, F4 contains 4 bits that specify one of the 16 operations performed in the ALU. Since no spare code is included, the ALU is active during the execution of every microinstruction. However, its activity is monitored by the rest of the machine through register Z which is loaded only when the Z_{in} signal is activated.

Grouping control signals into fields requires a little more hardware because decoding circuits must be used to decode the bit pattern of each field into individual control

signals. The cost of this additional hardware is more than offset by the reduced number of bits in each microinstruction, which results in a smaller control store. In Figure 8.5, only 20 bits are needed to store the patterns for the 42 signals.

So far, we have considered grouping and encoding only mutually exclusive control signals. We can extend this idea by enumerating the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code that represents the microinstruction. Such full encoding is likely to further reduce the length of microwords but also to increase the complexity of the required decoder circuits.

Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a *vertical organization*. On the other hand, the minimally encoded scheme of Figure 8.1 in which many resources can be controlled with a single microinstruction is called a *horizontal organization*. The horizontal approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources. The vertical approach results in considerably slower operating speeds because more microinstructions are needed to perform the desired control functions. Although fewer bits are required to each microinstruction, this does not imply that the total number of bits in the control store is smaller. The significant factor is that less hardware is needed to handle the execution of microinstructions.

Horizontal and vertical organizations represent the two organizational extremes in microprogrammed control. Many intermediate schemes are also possible in which the degree of encoding is a design parameter. The layout in Figure 8.5 is a horizontal organization because it groups only mutually exclusive microoperations in the same fields. As a result, it does not limit in any way the processor's ability to perform various microoperations in parallel.

8.4 MICROPROGRAMMING SEQUENCING

The simple microprogram example in Figure 8.1 requires only straightforward sequential execution of microinstructions except for the branch at the end of fetch phase. If each machine instruction is implemented by a microcontrol structure suggested in Figure 8.4 in which μ PC governs the sequencing would be sufficient. A microroutine is entered by decoding the machine instruction into a starting address that is loaded into the μ PC. Some branching capability within the microprogram can be introduced through special branch microinstructions that specify the branch address similar to the way branching is done in machine-level instructions.

With this approach, writing microprograms is fairly simple because standard software techniques can be used. However, this advantage is countered by two major disadvantages. Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store. If most machine instructions involve several addressing modes, there can be many instruction and addressing mode combinations. A separate microroutine for each of these combinations would produce considerable duplication of common parts. We want to organize the microprogram so that the microroutines share as many common parts as possible. This requires many branch microinstructions to transfer control among the various parts. Hence, a second disadvantage arises – execution time is longer because it takes more time to carry out the required branches.

Consider a more complicated example of a complete machine instruction. In earlier topic, we used instructions of the type

Add src, Rdst

which adds the source operand to the contents of register Rdst and places the sum in Rdst, the destination register. Let us assume that the source operand can be specified in the following addressing modes; register, autoincrement, autodecrement and indexed as well as the indirect forms of these four modes. We now use this instruction in conjunction with the processor structure in Figure 5.1 to demonstrate a possible microprogrammed implementation.

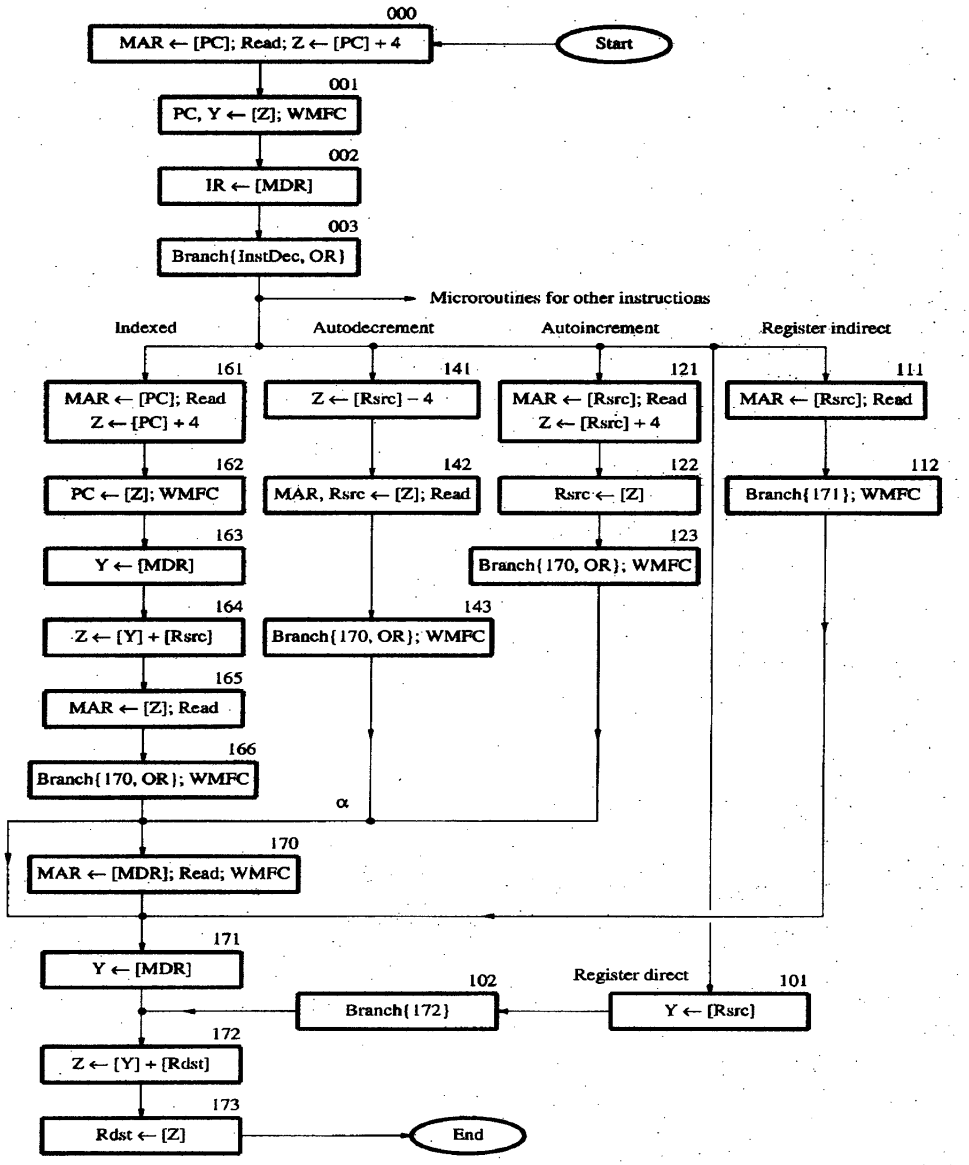


Figure 8.6 Flowchart of a microprogram for the Add sec, Rdst instruction.

A suitable microprogram is presented in flowchart form, for easier understanding in Figure 8.6. Each box in the flow chart corresponds to a microinstruction that controls the transfer and operations indicated within the box. The microinstruction is located at the address indicated by the octal number above the upper right-hand corner of the box. Each octal digit represents three bits. We use the octal notation in this example as a convenient shorthand notation for binary numbers.

Branch Address Modification Using Bit-Oring

The microprogram in Figure 8.6 shows that branches are not always made to a single branch address. This is a direct consequence of combining simple microroutines by sharing common parts. Consider the point labeled *a* in the figure. At this point, it is necessary to choose between actions required by direct and indirect addressing modes. If the indirect mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory. If the direct mode is specified, this fetch must be bypassed by branching immediately to location 171. The most efficient way to bypass microinstruction 170 is to have the preceding branch microinstructions specify the address 170 and then use an OR gate to change the least-significant bit of this address to 1 if the direct addressing mode is involved. This is known as the *bit-ORing* technique for modifying branch addresses.

An alternative to the bit-ORing approach is to use two conditional branch microinstructions at locations 123, 143, and 166. Another possibility is to include two next address fields within a branch microinstruction, one for the direct and one for the indirect address modes. Both of these alternatives are inferior to the bit-ORing technique.

Wide-Branch Addressing

The Figure 8.6 includes a wide branch in the microinstruction at location 003. The instruction decoder, abbreviated Inst Dec in the figure, generates the starting address of the microroutine that implements the instruction that has just been loaded into the IR. In our example, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. However, this address cannot be loaded as it is into the microprogram counter.

The source operand of the Add instruction can be specified in any of several addressing modes. The figure shows five possible branches that the add instruction may follow. From left to right these are the indexed, autodecrement, autoincrement, register direct, and register indirect addressing modes. The bit-ORing technique described above

can be used at this point to modify the starting address generated by the instruction decoder to reach the appropriate path. For the address shown in the figure, bit-ORing should change the address 101 to one of the five possible address values 161, 141, 121, 101, or 111, depending on the addressing mode used in the instruction.

Use of WMFC

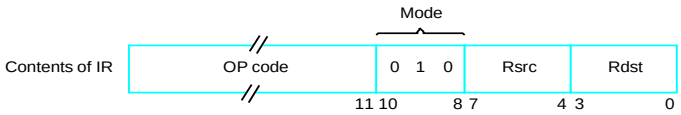
We have assumed that it is possible to issue a wait for MFC command in a branch microinstruction. This is done in the microinstruction at location 112, which causes a branch to the microinstruction in location 171. Combining these two operations introduces a subtle problem. The WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, the WMFC signal must inhibit any change in the contents of the microprogram counter during the waiting period.

Detailed Examination

Let us examine one path of the flowchart in figure 8.6 in more detail. Consider the case in which the source operand is accessed in the autoincrement mode. This is the path needed to execute the instruction

Add (Rsrc)+, Rdst

where Rsrc and Rdst are general putpose registers in the machine. Figure 8.7 shows the complete microroutine for fetching and executing this instruction. We assume that the instruction has a 3-bit field used to specify the addressing mode for the source operand, as shown. Bit patterns 11, 10, 01, and 00 located in bit 10 and 9, denote the indexed, autodecrement, autoincrement and register modes, respectively. For each of these modes, bit 8 is used to specify the indirect version. For example, 010 in the mode field specify the direct version of the autoincrement mode, whereas 011 specify the indirect version. We also assume that the processor has 16 registers that can be used for addressing purposes each specified using a 4-bit code. Thus, the source operand is fully specified using the mode field and the register indicated by bits 7 through 4. The destination operand is in the register by bits 3 through 0.



Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {$\mu PC \leftarrow$ 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, inZ
122	Z_{out}, $Rsrc_{in}$
123	μBranch {$\mu PC \leftarrow$ 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 8.7 Microinstruction for Add (Rsrc)+, Rdst.

Since any of the 16 general-purpose registers may be involved in determining the source and destination operand locations, the microinstructions refer to the respective control signals only as $Rsrc_{out}$, $Rdsst_{out}$, and $Rdst_{in}$. These signals must be translated into specific register transfer signals by the decoding circuitry connected to the Rsrc and Rdst address fields of the IR. This means that there are two stages of decoding. First, the microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved. The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating signals for the actual registers R0 to R15.

The microprogram in Figure 8.6 has been derived by combining the microroutines for all possible values in the mode field, resulting in a structure that requires many branch points. The example in Figure 8.7 has two branch points, so two branch microinstructions

are required. In each case, the expression in brackets indicates the branch address that is to be loaded into the μPC and how this address is modified using the bit-ORing scheme. Consider the microinstruction at location 123 as an example. Its unmodified version causes another fetch from the main memory corresponding to an indirect addressing mode. For a direct addressing mode, this fetch is bypassed by ORing the inverse of the indirect bit in src address field (bit 8 in the IR) with the 0 bit position of the μPC.

Another example of the use of ORing is the microinstruction in location 003. There are five starting addresses for the microroutine that implements the Add instruction in question, depending on the address mode specified for the source operand.

These addresses differ in the middle octal digit only. Hence, the required branch is implemented by using bit-ORing to modify octal digit of the pattern 101 obtained from the instruction decoder. The 3 bits to be ORed with this digit are supplied by the decoding circuitry connected to the src address mode field (bits 8, 9, and 10 of the IR). Microinstruction addresses have been chosen to make this modification easy to implement bits 4 and 5 of the μ PC are set directly from bits 9 and 10 in the IR. This suffices to select the appropriate microinstruction for all src address modes except one. The register indirect mode is covered by setting bit 3 of the μPC to 1 when $[\overline{IR}_{10}] \cdot [\overline{IR}_{9}] \cdot [IR_8]$ is equal to 1. Register indirect is a special case because it is only indirect mode that does not use the microinstruction at 170.

8.5 MICROINSTRUCTIONS WITH NEXT-ADDRESS FIELD

The microprogram in Figure 8.5 requires several branch microinstructions. These microinstructions perform no useful operation in the datapath; they are needed only to determine the address of the next microinstruction. Thus, they detract from the operating speed of the computer. The situation can become significantly worse when other microroutines are considered. The increase in branch microinstructions stems partly from limitations in the ability to assign successive addresses to all microinstructions that are generally executed in consecutive order.

This problem prompts us to reevaluate the sequencing technique built around an incrementable μ PC. A powerful alternative is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. This means, in effect, that every microinstruction becomes a branch microinstruction, in addition to its other functions.

The flexibility of this approach comes at the expense of additional bits for the address field. The severity of this penalty can be assessed as follows; in a typical computer, it is possible to design a complete microprogram with fewer than 4K microinstructions, employing perhaps 50 to 80 bits per microinstruction. This implies that an address field of 12 bits is required. Therefore, approximately one-sixth of the control store capacity would be devoted to addressing. Even if more extensive microprograms are needed, the address field would be only slightly larger.

The most obvious advantage of this approach is that separate branch microinstructions are virtually eliminated. Furthermore, there are few limitations in assigning addresses to microinstructions. These advantages more than offset any negative attributes and make the scheme very attractive. Since each instruction contains the address of the next instruction, there is no need for a counter to keep track of sequential addresses. Hence, the μ PC is replaced with a *microinstruction address register* (μAR) which is loaded from the next-address field in each microinstruction. A new control structure that incorporates this feature and supports bit-ORing is shown in Figure 8.8. The next-address bits are fed through the OR gates to the μAR so that the address can be modified on the basis of the data in the IR, external inputs, and condition codes. The decoding circuits generate the starting address of a given microroutine on the basis of the OP code in the IR.

Figure 8.8 Microinstruction-sequencing organization

Let us now reconsider the example of Figure 8.7 using the Microprogrammed control structure of Figure 8.8. We need several control signals that are not included in the microinstruction format in Figure 8.5. Instead of referring to registers R0 to R15 explicitly, we use the names Rsrc and Rdst which can be decoded into actual control signals with the data in the src and dst fields of the IR. Branching with the bit-ORing technique requires that we include the appropriate commands in the microinstructions. In the flowchart of Figure 8.6, bit-ORing is needed in microinstruction 003 to determine the address of the next microinstruction based on the addressing mode of the source operand. The addressing mode is indicated by bits 8 through 10 of the instruction register, as shown in Figure 8.6. Let the signal OR_{mode} control whether or not this bit-ORing is used. In microinstructions 123, 143, and 166, bit-ORing is used to decide indirect addressing of the source operand that is to be used. We use the signal OR_{index} for this purpose.

Figure 8.9 Format for microinstructions in the example of this Section

For simplicity, we use separate bits in the microinstructions to denote these signals. One bit in the microinstruction is used to indicate when the output of the instruction decoder is to be gated into the μ AR. Finally, each microinstruction contains an 8-bit field that holds the address of the next microinstruction. Figure 8.9 shows a complete format for these microinstructions.

Figure 8.10 Implementation of the microroutine of Figure 8. 7 using next-microinstruction address filed

This format is an expansion of the format in Figure 8.5. Using such microinstructions, we can implement the Microroutine of Figure 8.7 as shown in Figure 8.10. The revised routine has one less microinstruction. The branch microinstruction at location 123 has been combined with the microinstruction immediately preceding it. When microinstruction sequencing is controlled by a μPC, the End signal is used to reset the μPC to point to the starting address of the microinstruction. That fetches the next machine instruction to be executed. In our example, this starting address is 000_8. However, the Microroutine in Figure 8.10 does not terminate by producing the End signal. In an organization such as this, the starting address is not specified by a resetting mechanism triggered by the End signal – instead, it is specified explicitly in the FO field.

Figure 8.11 Details of the circuitry that generates the control signals in Figure 8.8

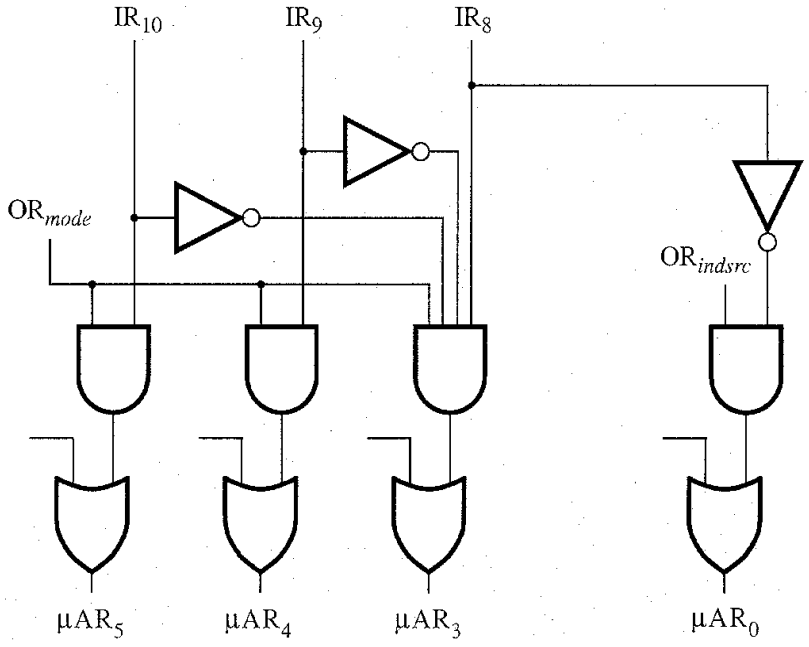


Figure 8.12 Control circuitry for bit-ORing

Figure 8.11 gives a more detailed diagram of the control structure of Figure 8.8. It shows how control signals can be decoded from the microinstruction fields and used to control sequencing. Detailed circuitry for bit-ORing is shown in Figure 8.12.

8.6 PREFETCHING MICROINSTRUCTIONS

One drawback of Microprogrammed control is that it leads to a slower operating speed because of the time it takes to fetch microinstructions from the control store. Faster operation is achieved if the next microinstruction is prefetched while the current one is being executed. In this way, the execution time can be overlapped with the fetch time. Prefetching microinstructions presents some organizational difficulties. Sometimes the status flags and the results of the currently executed microinstruction are needed to determine the address of the next microinstruction. Thus, straightforward prefetching occasionally prefetches a wrong microinstruction. In these cases, the fetch must be repeated with the correct address, which requires more complex hardware. However, the disadvantages are minor and the prefetching technique is often used.

8.7 EMULATION

The main function of Microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instructions. However, it also offers other interesting possibilities. Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented. Given a computer with a certain instruction set, it is possible to define additional machine instructions and implement them with extra microroutines.

An extension of the preceding idea leads to another interesting possibility. Suppose, we add to the instruction repertoire of a given computer, M_1, an entirely new set of instructions that is in fact the instruction set of a different computer, M_2. Programs written in the machine language of M_2 can then be run on Computer M_1, that is M_1 *emulates* M_2. Emulation allows us to replace obsolete equipment with more up-to-date

machines. If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs. Thus, emulation facilitates transitions to new computer systems with minimal disruption.

Emulation is easiest when the machines involved have similar architectures. However, emulation can also succeed using machines with totally different architectures.

Check your progress:

1. What is a microinstruction?
2. What is emulation?
3. What do you mean by bit-ORing?
4. Explain wide-branch addressing.
5. What is prefetching?

8.8 SUMMARY

In this unit, we have learnt Microprogrammed Control, Microinstructions and Microprogram Sequencing, Wide Branch Addressing, Microinstructions with Next-Address Field, Prefetching Microinstructions and Emulation.

8.9 KEY WORDS

CW: A *control word* (CW) is a word whose individual bits represent the various control signals.

Control store: The micro-routines for all instructions in the instruction set of a computer are stored in a special memory called the *control store.*

8.10 ANSWERS TO CHECK YOUR PROGRESS

1 8.2

2 8.7

3 8.4

4 8.4

5 8.6

8.11 UNIT-END EXERCISES AND ANSWERS

1. Elucidate Microprogrammed control.
2. Explain program sequencing.
3. Explain A Microinstruction with Next-Address Field.
4. Discuss about a *vertical organization and* a *horizontal organization.*

Answer: See

1 8.2

2 8.4

3 8.5

4 8.3

8.12 SUGGESTED READINGS

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: **Computer Organization**, 5th Edition, TMH 2002
2. William Stallings: **Computer Organization and Architecture,** 7th Edition, PHI 2006
3. Vincenet P. Heuring and Harry F. Jordan: **Computer Systems Design and Architecture,** 2nd Edition, Pearson Education, 2004.

UNIT 9: INTRODUCTION TO INPUT/OUTPUT DEVICES

Structure

9.0 OBJECTIVES

After studying this unit, you will be able to

- Understand how program-controlled I/O is performed using polling.
- NMB
- NMBMN
- BMNBMN
- GFGHF

9.1 INTRODUCTION

One of the basic features of a computer is its ability to exchange data with other devices. This communication capability enables a human operator, for example, to use a keyboard and a display screen to process text and graphics. We make extensive use of

computers to communicate with other computers over the Internet and access information around the globe. In other applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, transportation systems, banking and point-of-sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal to be sent to a speaker or a digitally coded command to change the speed of a motor, open a valve, or cause a robot to move in a specified manner. In short, a general-purpose computer should have the ability to exchange information with a wide range of devices in varying environments.

9.2 ACCESSING I/O DEVICES

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement, as shown in Figure 9.1. The bus enables all the devices connected to it to exchange information. Typically, it consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. When I/O devices and the memory share the same address space, the arrangement is called memory-mapped I/O.

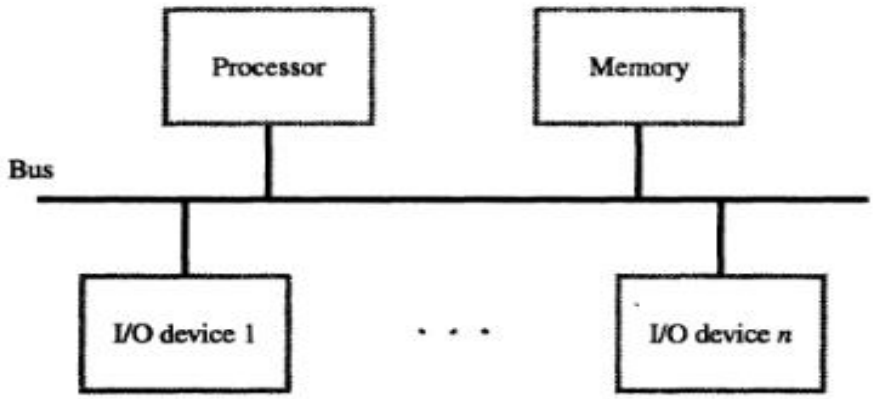


Figure 9.1: A single-bus structure

With memory-mapped I/O, any machine instruction that accesses memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of the input buffer associated with the keyboard, the instruction

Move DATAIN, RO

reads the data from DATAIN and stores them into processor register RO. Similarly, the instruction

Move RO, DATAOUT

sends the contents of register RO to location DATAOUT, which may be the output data buffer of a display unit or a printer.

Most computer systems use memory-mapped I/O. Some processors have special In and Out instructions to perform I/O transfers. For example, processors in the Intel have special I/O instructions and a separate 16-bit address space for I/O devices. When building a computer system based on these processors, the designer has the option of connecting I/O devices to use the special I/O address space or simply incorporating them as part of the memory address space. The latter approach is by far the most common as it leads to simpler software. One advantage of a separate I/O address space is that I/O devices deal with fewer address lines. Note that a separate I/O address space does not necessarily mean that the I/O address lines are physically separate from the memory address lines. A special signal on the bus indicates that the requested read or write transfer is an I/O operation. When this signal is asserted, the memory unit ignores the requested transfer. The I/O devices examine the low-order bits of the address bus to determine whether they should respond.

Figure 9.2 illustrates the hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of

executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.

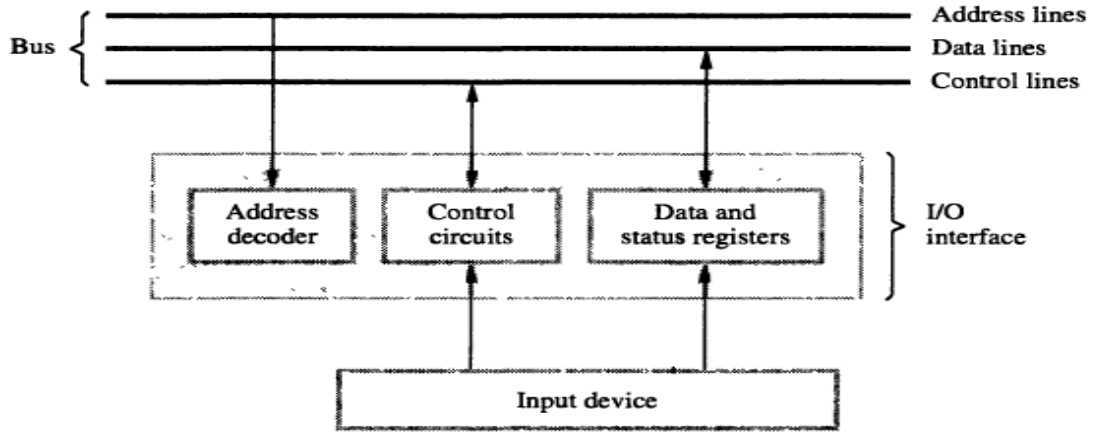


Figure 9.2: I/O interface for an input device

9.3 MEMORY MAPPED I/O DEVICES

Memory-mapped I/O uses the same address bus to address both memory and I/O devices, the memory and registers of the I/O devices are mapped to address values, as shown in Figure 1.3. So, when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O devices. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation might be temporary.

One merit of memory-mapped I/O is that, by discarding the extra complexity that port I/O brings, a CPU requires less internal logic and is thus cheaper, faster, easier to build, consumes less power and can be physically smaller; this follows the basic tenets

of reduced instruction set computing, and is also advantageous in embedded systems. The other advantage is that, because regular memory instructions are used to address devices, all of the CPU's addressing modes are available for the I/O as well as the memory, and instructions that perform an ALU operation directly on a memory operand — loading an operand from a memory location, storing the result to a memory location, or both, can be used with I/O device registers as well. In contrast, port-mapped I/O instructions are often very limited, often providing only for simple load and store operations between CPU registers and I/O ports, so that, for example, to add a constant to a port-mapped device register would require three instructions: read the port to a CPU register, add the constant to the CPU register, and write the result back to the port.

As 16-bit processors have become obsolete and replaced with 32-bit and 64-bit in general use, reserving ranges of memory address space for I/O is less of a problem, as the memory address space of the processor is usually much larger than the required space for all memory and I/O devices in a system. Therefore, it has become more frequently practical to take advantage of the benefits of memory-mapped I/O.

Memory-mapped I/O is preferred in x86-based architectures because the instructions that perform port-based I/O are limited to one register: EAX, AX, and AL are the only registers that data can be moved in to or out of, and either a byte-sized immediate value in the instruction or a value in register DX determines which port is the source or destination port of the transfer. Since any general purpose register can send or receive data to or from memory and memory-mapped I/O, memory-mapped I/O uses less instructions and can run faster than port I/O. AMD did not extend the port I/O instructions when defining the x86-64 architecture to support 64-bit ports, so 64-bit transfers cannot be performed using port I/O.

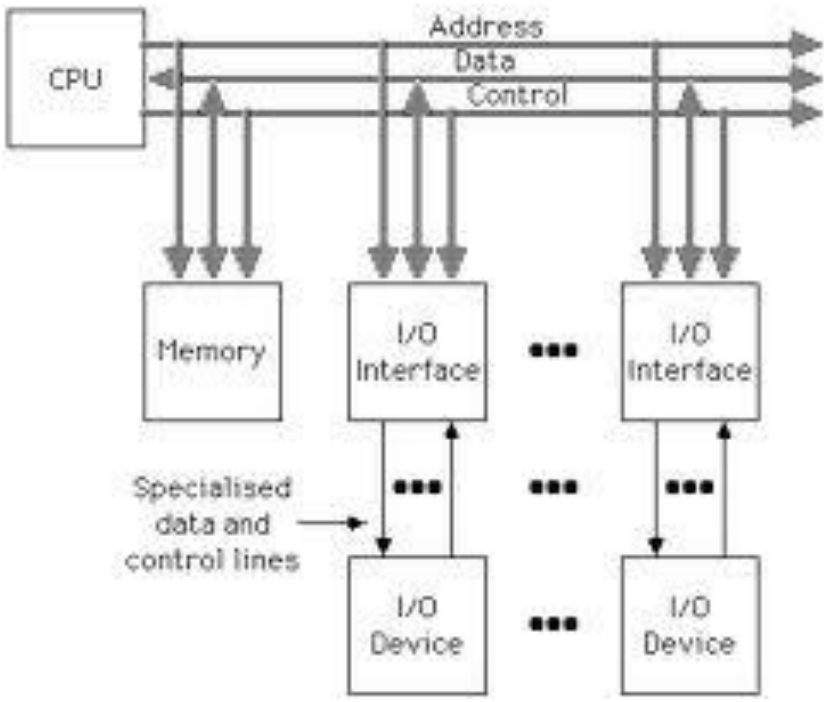


Figure 1.3: Memory mapped I/O device Structure

9.4 I/O MAPPED I/O DEVICES

I/O mapped I/O devices is also known as port mapped I/O devices. Port-mapped I/O often uses a special class of CPU instructions specifically for performing I/O. This is found on Intel microprocessors, with the IN and OUT instructions. These instructions can read and write one to four bytes to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

A device's direct memory access (DMA) is not affected by CPU-to-device communication methods, like memory mapping. This is because by definition DMA is a memory-to-device communication method that bypasses the CPU.

9.5 I/O INTERFACING FOR AN INPUT DEVICE

The I/O interface consists of the circuitry required to transfer data between the computer bus and an I/O device. Therefore, on one side of the interface we have the bus signals for address, data, and control. On the other side we have a data path with its associated controls, which enables transfer of data between the interface and an I/O device. This side is device-dependent. However, it can be classified as either a parallel or a serial interface. A parallel interface transfers data in the form of one or more bytes simultaneously to or from the device. A serial interface transmits and receives data one bit at a time. Communication with the bus is the same for both formats; the conversion from parallel to the serial format, and vice versa, takes place inside the interface circuit.

9.6 MECHANISMS FOR INTERFACING I/O DEVICES

Parallel Interface

Figure 9.4 shows a scheme for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character.

The output from the encoder consists of the bits that represent an encoded character and one control signal, called **Valid**, which indicates that a key is being pressed. This information is sent to the input interface. The interface contains data register, DATAIN, and a status flag, SIN. When a key is pressed, the **Valid** signal changes from 0 to 1. This causes the ASCII code to be loaded into the DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register. The I/O interface is connected to the asynchronous bus on which transfers are controlled using the handshake signals **Ready** and **Accept**, as indicated in Figure 9.4. The third control line, R/W distinguishes read and write transfers.

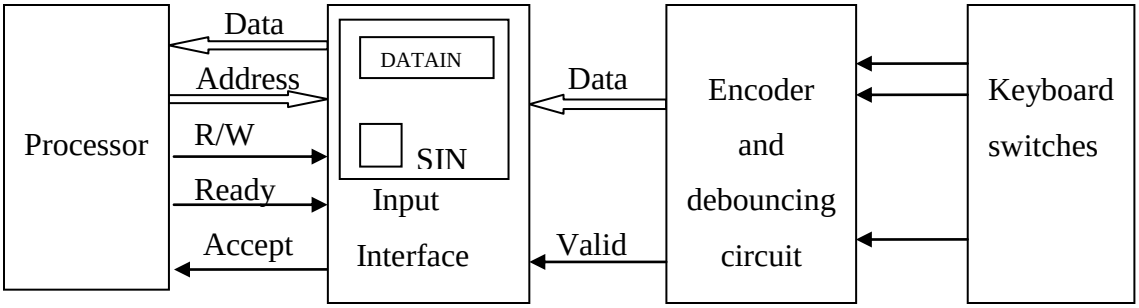


Figure 9.4 Keyboard connected to processor

Serial Interface

A serial interface is used to connect the processor to I/O devices that require transmission of data one bit at a time. The key feature of serial interface is a circuit capable of communicating in bit-serial fashion on the device side and in bit-parallel fashion on the bus side. Transformation between parallel and serial formats is achieved with shift registers that have parallel access capability. As shown in Figure 9.5, it includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are loaded into the output shift register, from which the bits are shifted out and sent to the I/O device.

The status flag SIN and SOUT are used as if SIN is set to 1 when new data are loaded into DATAIN; it is cleared to 0 when processor reads the contents of DATAIN. The SOUT flag indicates whether the output buffer is available. It is cleared to 0 when the processor writes new data into the DATAOUT register and set to 1 when data are transferred from DATAOUT into the output shift register.

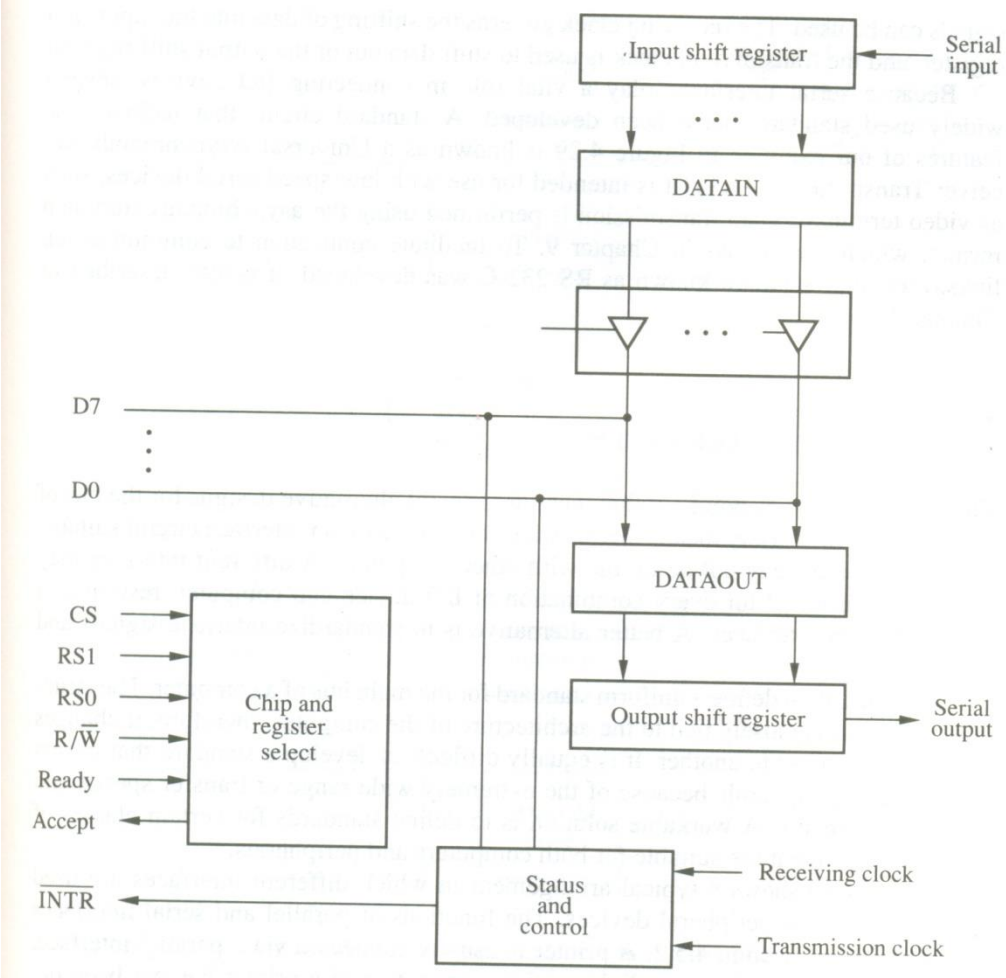


Fig 9.5 A Serial interface

DMA

Implemented with special controller that transfers data between memory and I/O device independent of the processor

Three steps in DMA transfers:

1. Processor sets up the DMA transfer by supplying identity of device, operation to perform, memory address that is source or destination of data, number of bytes to be transferred.

2. DMA controller starts the operation (arbitrates for the bus, supplies address, reads or writes data), until the entire block is transferred.

3. DMA controller interrupts the processor, which then takes the necessary actions.

Check your progress:

1 What is DMA?

2 What is serial interface.

3 Explain I/O interfacing for an input device.

9.7 SUMMARY

In this unit, we have discussed the three basic approaches for I/O transfers. The simplest technique is programmed I/O in which the processor performs all the necessary control functions under direct control of program instructions.

9.8 KEYWORDS

Interface

I / O Device

I / O transfers

DMA: Direct Memory Access

9.9 ANSWERS TO CHECK YOUR PROGRESS

1 9.6

2 9.6

3 9.5

9.9 UNIT END EXERCISES AND ANSWERS

1. Explain access memory I/O devices.

2. Differentiate between Memory mapped I/O devices and I/O mapped I/O devices
3. Explain the mechanism for interfacing I/O devices.

Answer: SEE

1 9.3

2 9.3 and 9.4

3 9.6

9.10 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.

Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI, 1992.

UNIT – 10: INTERRUPTS

Structure

10.0 OBJECTIVES

After studying this unit, you should be able to:

- Explain how to generate the appropriate timing signals required by the bus control scheme.
- Explain how the processor can recognize the device requesting an interrupt.
- Elucidate how different devices are likely to require different interrupt -service routines.
- Discuss how can the processor obtain the starting address of the appropriate routine in each case?

- Explain if a device is to be allowed to interrupt the processor while another interrupt is being serviced?
- Discuss how should two or more simultaneous interrupt requests be handled?

10.1 INTRODUCTION

Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time.

10.2 INTERRUPT I/O

Let us consider a situation when a program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an interrupt to the processor. At least one of the bus control lines, called an interrupt-request line, is usually dedicated for this purpose. Since the processor is no longer required to continuously check the status of external devices; it can use the waiting period to perform other useful functions. Indeed, by using interrupts, such waiting periods can ideally be eliminated.

Consider a task that requires some computations to be performed and the results to be printed on a line printer. This is followed by more computations and output, and so on. Let the program consist of two routines, COMPUTE and PRINT. Assume that COMPUTE produces a set of n lines of output, to be printed by the PRINT routine.

The required task may be performed by repeatedly executing first the COMPUTE routine and then the PRINT routine. The printer accepts only one line of text at a time. Hence, the PRINT routine must send one line of text, wait for it to be printed, and then send the next line, and so on, until all the results have been printed. The disadvantage of this simple approach is that the processor spends a considerable amount of time waiting for the printer to become ready. If it is possible to overlap printing and computation, that is, to execute the COMPUTE routine while printing is in progress, a faster overall speed of execution will result. This may be achieved as follows. First, the COMPUTE routine is executed to produce the first n lines of output. Then, the PRINT routine is executed to send the first line of text to the printer. At this point, instead of waiting for the line to be printed the PRINT routine may be temporarily suspended and execution of the COMPUTE routine continued. Whenever the printer becomes ready, it alerts the processor by sending an interrupt-request signal. In response, the processor interrupts execution of the COMPUTE routine and transfers control to the PRINT routine. The PRINT routine sends the second line to the printer and is again suspended. Then the interrupted COMPUTE routine resumes execution at the point of interruption. This process continues until all n lines have been printed and the PRINT routine ends.

The PRINT routine will be restarted whenever the next set of n lines is available for printing. If COMPUTE takes longer to generate n lines than the time required to print them, the processor will be performing useful computations all the time.

This example illustrates the concept of interrupts. The routine executed in response to an interrupt request is called the interrupt-service routine, which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction in Figure 10.1.

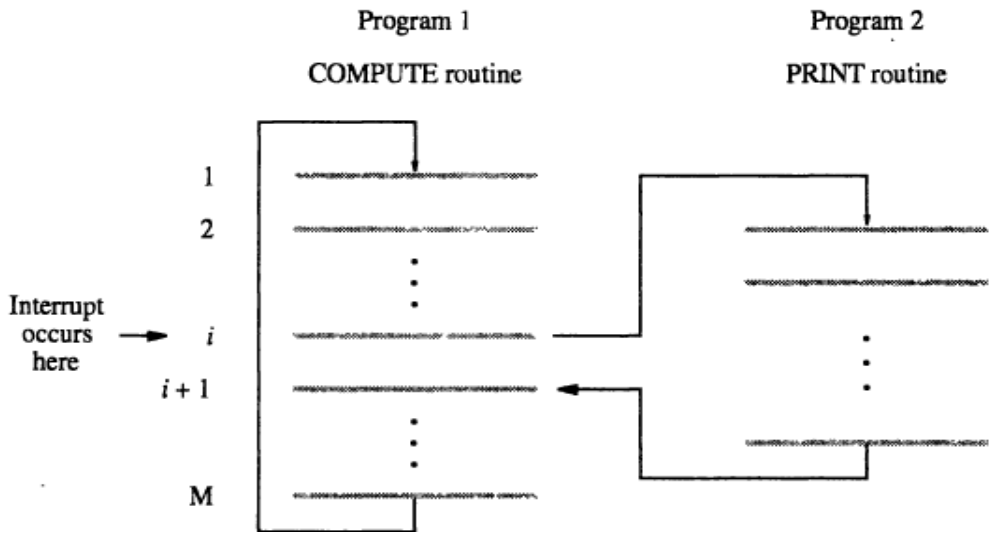


Figure 10.1: Transfer of control through the use of interrupts

The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction i + 1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i + 1, must be put in temporary storage in a known location. A Return – from intenupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction i + 1. In many processors, the return address is saved on the processor stack. Alternatively, it may be saved in a special location, such as a register provided for this purpose.

We should note that as part of handling interrupts, the processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus. An interrupt-acknowledge signal, used in some of the interrupt schemes to be discussed later, serves this function. A common alternative is to have the transfer of data between the processor and the I/O device interface accomplish the same purpose. The execution of an

instruction in the interrupt -service routine that accesses a status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.

We pointed out that an I/O device requests an interrupt by activating a bus line called interrupt-request. Most computers are likely to have several I/O devices that can request n interrupts. A single interrupt -request line may be used to serve n devices as depicted in Figure 9.2. All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to V_{dd}. This is the inactive state of the line. When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt-request signal, INTR, received by the processor to go to 1.

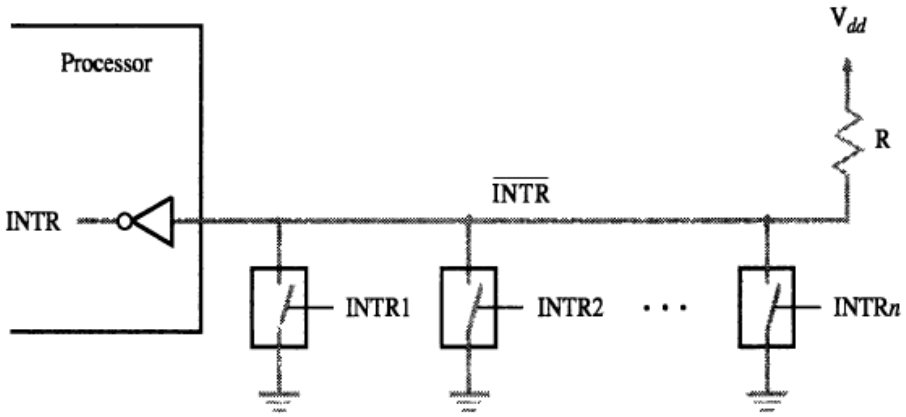


Figure 10.2: An equivalent circuit for an open-drain bus used to implement a common Interrupt-request line

Since the closing of one or more switches will cause the line voltage to drop to 0, the value of INTR is the logical OR of the requests from individual devices, that is,
$INTR = INTR_1 + ... + INTR_n$

It is customary to use the complemented form, INTR, to name the interrupt-request signal on the common line, because this signal is active in the low-voltage state.

10.3 ENABLING AND DISABLING INTERRUPTS

The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another. Because interrupts can arrive at any time, they may alter the sequence of events from that envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled. A fundamental facility found in all computers is the ability to enable and disable such interruptions as desired.

There are many situations in which the processor should ignore interrupt requests. For example, in the case of the Compute-Print program of Figure 10.1, an interrupt request from the printer should be accepted only if there are output lines to be printed. After printing the last line of a set of n lines, interrupts should be disabled until another set becomes available for printing. In another case, it may be necessary to guarantee that a particular sequence of instructions is executed to the end without interruption because the interrupt-service routine may change some of the data used by the instructions in question. For these reasons, some means for enabling and disabling interrupts must be available to the programmer. A simple way is to provide machine instructions, such as Interrupt-enable and Interrupt-disable that perform these functions.

10.4 HANDLING MULTIPLE DEVICES:

When a request is received over the common interrupt-request line in Figure 9.2, additional information is needed to identify the particular device that activated the line. Furthermore, if two devices have activated the line at the same time, it must be possible to break the tie and select one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced. The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it

sets to 1 one of the bits in its status register, which we will call the Interrupt Request (IRQ) bit. For example, bits KIRQ and DIRQ in Figure 9.2., are the interrupt request bits for the keyboard and the display, respectively. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The first device encountered with its IRQ bit set is the device that should be serviced. An appropriate subroutine is called to provide the requested service.

The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of all the devices that may not be requesting any service. An alternative approach is to use vectored interrupts.

10.5 VECTORED INTERRUPTS:

To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term vectored interrupts refers to all interrupt -handling schemes based on this approach.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the starting address of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine. In many computers, this is done automatically by the interrupt-handling mechanism. The location pointed to by the interrupting device is used to store the starting address of the interrupt-service routine. The processor reads this address, called the interrupt vector, and

loads it into the PC. The interrupt vector may also include a new value for the processor status register.

10. 6 INTERRUPTS NESTING:

We suggested in earlier Section that interrupts should be disabled during the execution of an interrupt – service routine, to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, once started, always continues to completion before the processor accepts an interrupt request from a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices.

For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation. Consider, for example, a computer that keeps track of the time of day using a real-time clock. This is a device that sends interrupt requests to the processor at regular intervals. For each of these requests, the processor executes a short interrupt-service routine to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on. Proper operation requires that the delay in responding to an interrupt request from the real-time clock be small in comparison with the interval between two successive requests. To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept an interrupt request from the clock during the execution of an interrupt-service routine for another device.

The above example suggests that I/O devices should be organized in a priority structure. An interrupt request from a high-priority device should be accepted while the processor is servicing another request from a lower-priority device. A multiple-level priority organization means that during execution of an interrupt-service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts

interrupts only from devices that have priorities higher than its own. At the tie the execution of an interrupt -service routine for some device is started, the priority of the processor is raised to that of the device. This action disables interrupts from devices at the same level of priority or lower. However, interrupt requests from higher-priority devices will continue to be accepted.

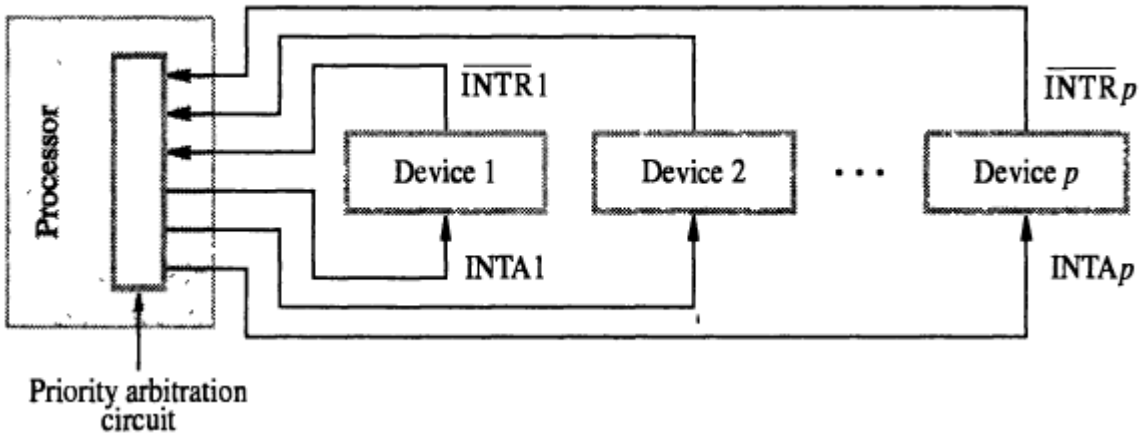


Figure 10.3: Implementation of interrupt priority using individual interrupt-request and acknowledge lines

A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device, as shown in Figure 10.3. Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

SIMULTANEOUS REQUESTS

Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which request to service first. Using a priority scheme such as that of Figure 10.4, the solution is straightforward. The processor simply accepts the request having the highest priority. If

several devices share one interrupt-request line, as in Figure 10.3, some other mechanism is needed.

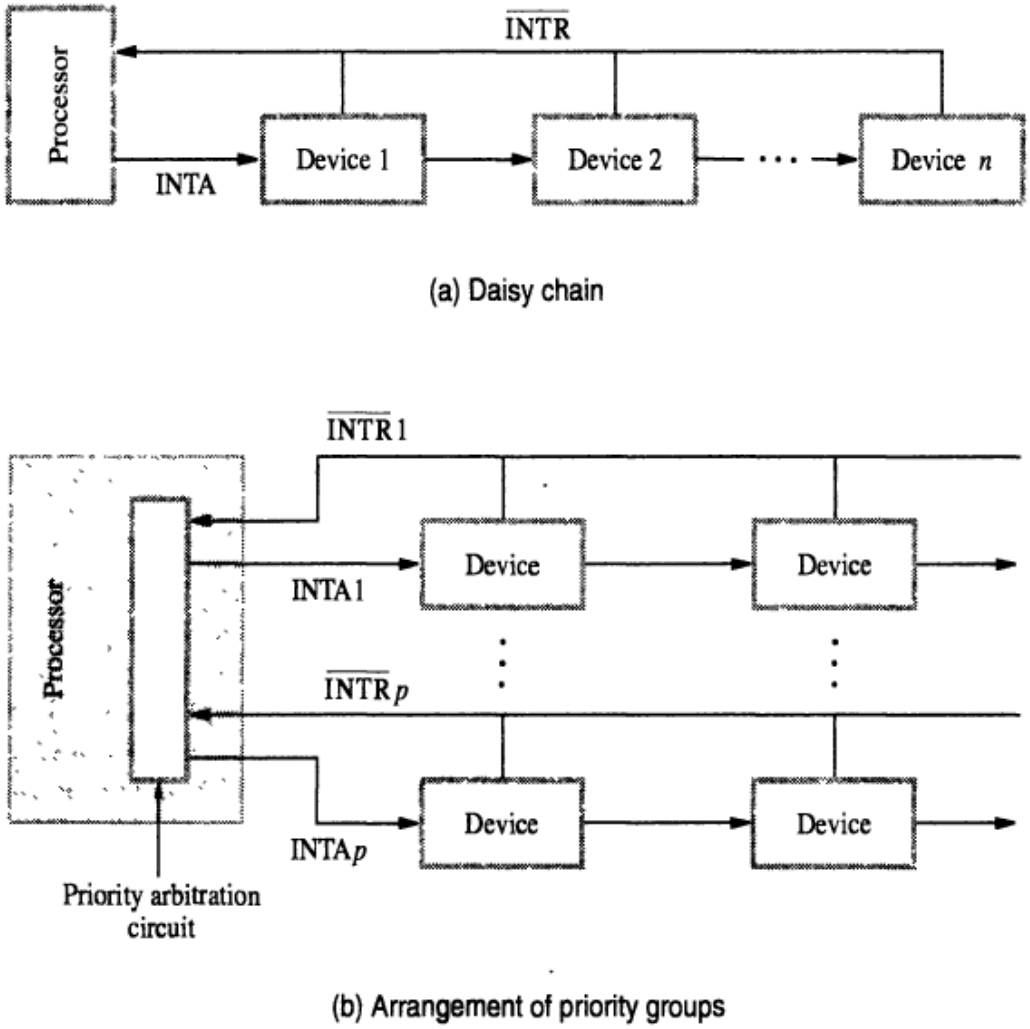


Figure 10.4: Interrupt priority schemes

Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a daisy chain, as shown in Figure 10.4 (a). The interrupt-request line INTR is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices

raise an interrupt request and the INTR line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.

The scheme in Figure 10.4(b) requires considerably fewer wires than the individual connections in Figure 10.3. The main advantage of the scheme in Figure 10.3 is that it allows the processor to accept interrupt requests from some devices but not from others, depending upon their priorities. The two schemes may be combined to produce the more general structure in Figure 10.4(b). Devices are organized in groups, and each group is connected at a different priority level. Within a group, devices are connected in a daisy chain. This organization is used in many computer systems.

10.7 PRIORITY STRUCTURES:

Note that the general organization in Fig 10.4 (b) makes it possible for a device to be connected to several priority levels. At any given time, the device requests an interrupt at a priority level consistent with the urgency of the service requested. This approach offers additional flexibility at the expense of more complex control circuitry in the device interface.

10.8 CONTROLLING DEVICE REQUESTS:

The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit. The keyboard interrupt-enable, KEN, and display interrupt-enable, DEN, flags in register CONTROL perform this function. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status

flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.

To summarize, there are two independent mechanisms for controlling interrupt requests. At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an interrupt request. At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted.

Check your progress:

1. What is an interrupt?

2. Briefly explain the different types of interrupts.

3. How can the processor recognize the device requesting an interrupt?

4. What are the functions of controlling device requests?

10.9 SUMMARY

The second approach for I/O operations is based on the use of interrupts; this mechanism makes it possible to interrupt normal execution of programs in order to service higher-priority requests that require more urgent attention. Although all computers have a mechanism for dealing with such situations the complexity and sophistication of interrupt-handling schemes vary from one computer to another.

10.10 KEYWORDS

Interrupt: a hardware signal

Vectored Interrupts

Interrupt Nesting.

10.11 ANSWER TO CHECK YOUR PROGRESS

1. **10.2**
2. **10.4 TO 10.7**
3. **10.2**
4. **10.8**

10.12 UNIT-END EXERCISES

1. How does an interrupt occur?
2. Discuss how one enables and disables interrupts.
3. How should two or more simultaneous interrupt requests be handled?
4. Distinguish between vectored interrupts and an interrupt nesting.
5. Explain the mechanism of interrupts.

Answer: SEE

1. 10.2
2. 10.3
3. 10.6
4. 10.5 and 10.6
5. 10.2

10.13 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.
Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI, 1992.

UNIT - 11: DIRECT MEMEORY ACCESS

Structure

11.0 OBJECTIVES

By studying this unit, you will be able to

- Understand exceptions.
- Explain the direct memory access as an I/O mechanism for high-speed devices.
- How data transfer over synchronous and asynchronous buses are performed.
- Discuss the design of I/O interface circuits which Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port.

11.1 INTRODUCTION

In the previous sections, we discussed about in the data transfer between the processor and I/O devices. Data are transferred by executing instructions such as

Move DATAIN, RO

An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready. To do this, the processor either polls a status flag in the device interface or waits for the device to send an interrupt request. In either case, considerable overhead is incurred, because several program instructions must be executed for each data word transferred. In addition to polling the status register of the device, instructions are needed for incrementing the memory address and keeping track of the word count. When interrupts are used, there is a additional overhead associated with saving and restoring the program counter and other state information.

To transfer large blocks of data at high speed, an alternative approach is used. A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called Direct Memory Access (DMA).

11.2 EXCEPTIONS:

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. So far, we have dealt only with interrupts caused by requests received during I/O data transfers. However, the interrupt mechanism is used in a number of other situations.

The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception. We now describe a few other kinds of exceptions.

RECOVERY FROM ERRORS:

Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt.

The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero.

When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine. This routine takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an I/O interrupt, the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error, execution of the interrupted instruction cannot usually be completed, and the processor begins exception processing immediately.

DEBUGGING:

Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer to find errors in a program. The debugger uses exceptions to provide two important facilities called trace and breakpoints.

When a processor is operating in trace mode, an exception occurs after execution of every instruction, using the debugging program as the exception-service routine. The debugging program enables the user to examine the contents of registers, memory locations, and so on. On return from the debugging program, the next instruction in the program being debugged is executed, and then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program.

Breakpoints provide a similar facility, except that the program being debugged is interrupted only at specific points selected by the user. An instruction called Trap or Software-interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. While debugging a program, the user may wish to interrupt program execution after instruction 'i'. The debugging routine saves instruction i + 1 and replaces it with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. This gives the user a chance to examine memory and register contents. When the user is ready to continue executing the program being debugged, the debugging routine restores the saved instruction that was at location i + 1 and executes a Return-from-interrupt instruction.

PRIVILEGE EXCEPTION:

To protect the operating system of a computer from being corrupted by user programs, certain instructions can be executed only while the processor is in the supervisor mode. These are called privileged instructions. For example, when the processor is running in the user mode, it will not execute an instruction that changes the priority level of the processor or that enables a user program to access areas in the computer memory that have been allocated to other users. An attempt to execute such an instruction will produce a privilege exception, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

11.3 DIRECT MEMORY ACCESS

DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a DMA controller. The DMA controller performs the functions that would normally be carried out by the processor when accessing the main memory. For each word transferred, it provides the memory address and all the bus signals that control data transfer. Since it has to transfer blocks of data, the DMA

controller must increment the memory address for successive words and keep track of the number of transfers.

Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor. To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the direction of the transfer. On receiving this information, the DMA controller proceeds to perform the requested operation. When the entire block has been transferred, the controller informs the processor by raising an interrupt signal. While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer. I/O operations are always performed by the operating system of the computer in response to a request from an application program. The OS is also responsible for suspending the execution of one program and starting another. Thus, for an I/O operation involving DMA, the OS puts the program that requested the transfer in the Blocked state, initiates the DMA operation, and starts the execution of another program. When the transfer is completed, the DMA controller informs the processor by sending an interrupt request. In response, the OS puts the suspended program in the Runnable state so that it can be selected by the scheduler to continue execution.

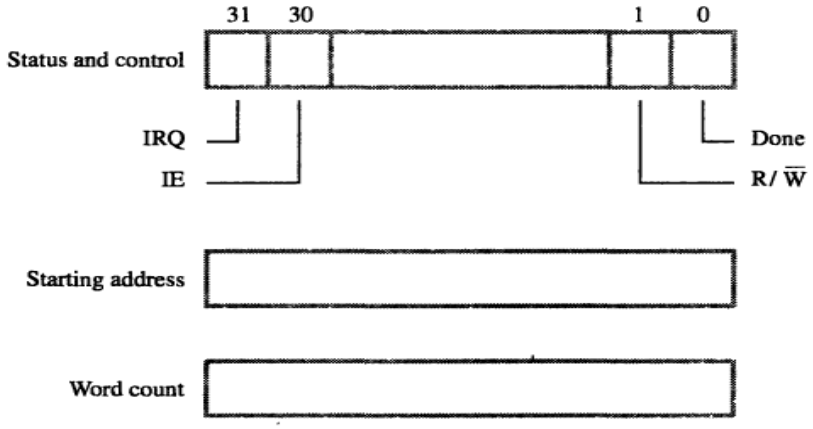


Figure 11.1: Registers in a DMA interface

Figure 11.1 shows an example of the DMA controller registers that are accessed by the processor to initiate transfer operations. Two registers are used for storing the

starting address and the word count. The third register contains status and control flags. The R/W bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a write operation. When the controller has completed transferring a block of data and is ready to receive another command, it sets the done flag to 1. Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt. .

11.4 DMA OPERATION

Direct memory access (DMA) is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory independently of the central processing unit (CPU).

Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the Read or Write operation, and is thus unavailable to perform other work. With DMA, the CPU initiates the transfer, does other operations while the transfer is in progress, and receives an interrupt from the DMA controller when the operation is done. This feature is useful any time the CPU cannot keep up with the rate of data transfer, or where the CPU needs to perform useful work while waiting for a relatively slow I/O data transfer. Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards and sound cards. DMA is also used for intra-chip data transfer in multi-core processors. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, allowing computation and data transfer to proceed in parallel.

DMA can also be used for "memory to memory" copying or moving of data within memory. DMA can offload expensive memory operations, such as large copies or scatter-gather operations, from the CPU to a dedicated DMA engine. An implementation example is the I/O Acceleration Technology.

Advantages of DMA

- Computer system performance is improved by direct transfer of data between memory and I/O devices, bypassing the CPU.
- CPU is free to perform operations that do not use system buses.

Disadvantages of DMA

- In case of Burst Mode data transfer, the CPU is rendered inactive for relatively long periods of time.

11.5 REGISTERS IN A DMA INTERFACE

The **DMA** controller includes three registers: an address register, a byte count register, and a control register.

- The address register contains 16 bits that specify the desired location in memory. The address bits go through a bus buffer into the address bus. The address register is incremented after each DMA byte transfer.
- The byte count register holds the number of bytes to be transferred. This register is decremented after each byte transfer and internally tested for zero.
- The control register specifies the mode of transfer-whether it is into (write) or out of (read) memory.

All registers in DMA appear to the microprocessor as an I/O interface. Thus, the processor can read from or write into the DMA registers under program control via the data bus.

11.6 USE OF DMA CONTROLLERS IN A COMPUTING SYSTEM

DMA transfer is very useful in many microcomputer system applications. It is useful for fast transfer of information between magnetic tape cassettes and system RAM. It is also useful for communication with interactive terminal systems having CRT screens or with television screens used for video games. Typically, an image of the screen display is kept in a memory which can be updated under processor control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

The potential application of DMA is in a multiprocessor system forming a network of two or more processors. Communication between processors can be maintained with a shared memory that can be accessed by all processors. DMA is a convenient method for transferring information between the common memory and the various processors in the network.

Check your progress:

1. Explain the term exception.
2. What is Direct Memory Access?
3. Explain the functions of debugging process.

11.7 SUMMARY

In this unit we briefly explained exceptions. We also discussed the third I/O scheme which involves direct memory access; the DMA controller that transfers data between an I/O device and the main memory without continuous processor intervention. Access to memory is shared between the DMA controller and the processor.

11.8 KEYWORDS

Exceptions

DMA

DMA Controller

Interface Circuits

11.9 ANSWER TO CHECK YOUR PROGRESS

1. **11.2**
2. **11.3**
3. **11.2**

11.10 UNIT-END EXERCISES AND ANSWERS

1. How does the system recover from errors?
2. Explain the functions of DMA interface.
3. What is Synchronous Bus? Explain.
4. Explain DMA operations.

Answer : SEE

1. 11.2
2. 11.3
3. 11.3
4. 11.4

11.11 SIGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.

Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI. 1992

UNIT–12: I/O HARDWARE AND STANDARD I/O INTERFACES

Structure

12.0 OBJECTIVES

After studying this unit, you will get a good picture:

- I/O hardware
- Commercial bus standards, in particular the PCI, SCSI buses.

12.1 INTRODUCTION

The previous sections point out that there are several alternative designs for the bus of a computer. This variety means that I/O devices fitted with an interface circuit suitable for one computer may not be usable with other computers. A different interface may have to be designed for every combination of I/O device and computer, resulting in

many different interfaces. The most practical solution is to develop standard interface signals and protocols.

It is helpful at this point to understand how a computer system is put together. A typical personal computer, for example, includes a printed circuit board called the motherboard. This board houses the processor chip, the main memory, and some I/O interfaces. It also has a few connectors into which additional interfaces can be plugged.

12.2 I/O HARDWARE

The processor, main memory, and I/O devices can be interconnected by means of a common bus whose primary function is to provide a communications path for the transfer of data. The bus includes the lines needed to support interrupts and arbitration. In this section, we discuss the main features of the bus protocols used for transferring data. A bus protocol is the set of rules that govern the behavior of various devices connected to the bus as to when to place information on the bus, assert control signals, and so on. After describing bus protocols, we will present examples of interface circuits that use these protocols.

The bus lines used for transferring data may be grouped into three types: data, address, and control lines. The control signals specify whether a read or a write operation is to be performed. Usually, a single R/W line is used. It specifies Read when set to 1 and Write when set to O. When several operand sizes are possible, such as byte, word, or long word, the required size of data is indicated.

The bus control signals also carry timing information. They specify the times at which the processor and the I/O devices may place data on the bus or receive data from the bus. A variety of schemes have been devised for the timing of data transfers over a bus. These can be broadly classified as either synchronous or asynchronous schemes.

SYNCHORONOUS BUS

In a synchronous bus, all devices derive timing information from a common clock line. Equally spaced pulses on this line define equal time intervals. In the simplest form of a synchronous bus, each of these intervals constitutes a bus cycle during which one

data transfer can take place. Such a scheme is illustrated in Figure 12.1 . The address and data lines in this and subsequent figures are shown as high and low at the same time. This is a common convention indicating that some lines are high and some low, depending on the particular address or data pattern being transmitted. The crossing points indicate the times at which these patterns change. A signal line in an indeterminate or high impedance state is represented by an intermediate level half-way between the low and high signal levels.

Let us consider the sequence of events during an input (read) operation. At time t_o, the master places the device address on the address lines and sends an appropriate command on the control lines. In this case, the command will indicate an input operation and specify the length of the operand to be read, if necessary. Information travels over the bus at a speed determined by its physical and electrical characteristics. The clock pulse width, $t_l - t_o$, must be longer than the maximum propagation delay between two devices connected to the bus. It also has to be long enough to allow all devices to decode the address and control signals so that the addressed device (the slave) can respond at time t_1. It is important that slaves take no action or place any data on the bus before t_l. The information on the bus is unreliable during the period of $t_o - t_1$, because signals are in changing state. The addressed slave places the requested input data on the data lines at time t_1.

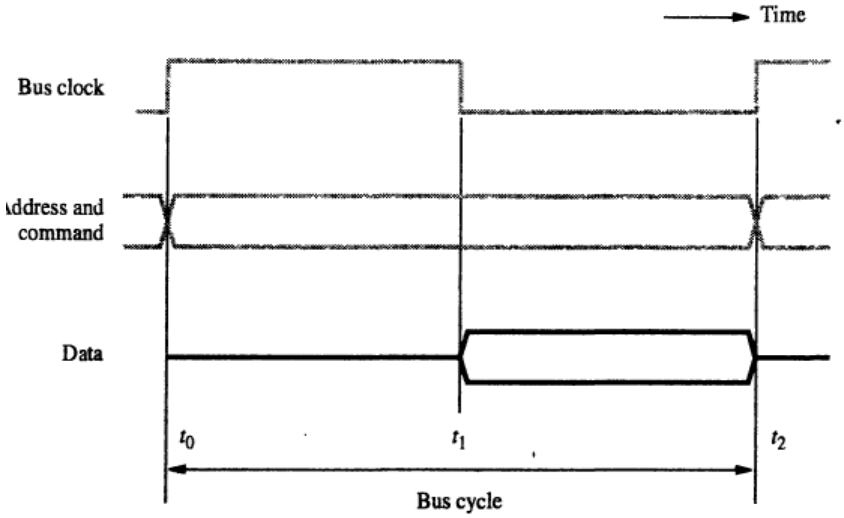


Figure 12.1: Timing of an input transfer on a synchronous bus

At the end of the clock cycle, at time t_2, the master strobes the data on the data lines into its input buffer. In this context, "strobe" means to capture the values of the data at a given instant and store them into a buffer. For data to be loaded correctly into any storage device, such as a register built with flip-flops, the data must be available at the input of that device for a period greater than the setup time of the device. Hence, the period $t_2 - t_1$ must be greater than the maximum propagation time on the bus plus the setup time of the input buffer register of the master.

A similar procedure is followed for an output operation. The master places the output data on the data lines when it transmits the address and command information. At time t_2, the addressed device strobes the data lines and loads the data into its data buffer.

The timing diagram in Figure 12.1 is an idealized representation of the actions that take place on the bus lines. The exact times at which signals actually change state are somewhat different from those shown because of propagation delays on bus wires and in the circuits of the devices.

MULTIPLE-CYCLE TRANSFERS:

The processor has no way of determining whether the addressed device as actually responded. It simply assumes that, at t_2, the output data have been received by the I/O device or the input data are available on the data lines. If, because of a malfunction, the device does not respond, the error will not be detected.

To overcome these limitations, most buses incorporate control signals that represent a response from the device. These signals inform the master that the slave has recognized its address and that it is ready to participate in a data-transfer operation. They also make it possible to adjust the duration of the data-transfer period to suit the needs of the participating devices. To simplify this process, a high-frequency clock signal is used such that a complete data transfer cycle would span several clock cycles. Then, the number of clock cycles involved can vary from one device to another.

An example of this approach is shown in Figure 12.2. During clock cycle 1, the master sends address and command information on the bus, requesting a read operation. The slave receives this information and decodes it. On the following active edge of the

clock, that is, at the beginning of clock cycle 2, it makes a decision to respond and begins to access the requested data. We have assumed that some delay is involved in getting the data, and hence the slave cannot respond immediately. The data become ready and are placed on the bus in clock cycle 3. At the same time, the slave asserts a control signal called Slave-ready. The master, which has been waiting for this signal, strobes the data into its input buffer at the end of clock cycle 3. The bus transfer operation is now complete, and the master may send a new address to start a new transfer in clock cycle 4.

The Slave-ready signal is an acknowledgment from the slave to the master, confirming that valid data have been sent. In the example in Figure 12.1, the slave responds in cycle 3. Another device may respond sooner or later. The Slave-ready signal allows the duration of a bus transfer to change from one device to another. If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, and then aborts the operation. This could be the result of an incorrect address or a device malfunction.

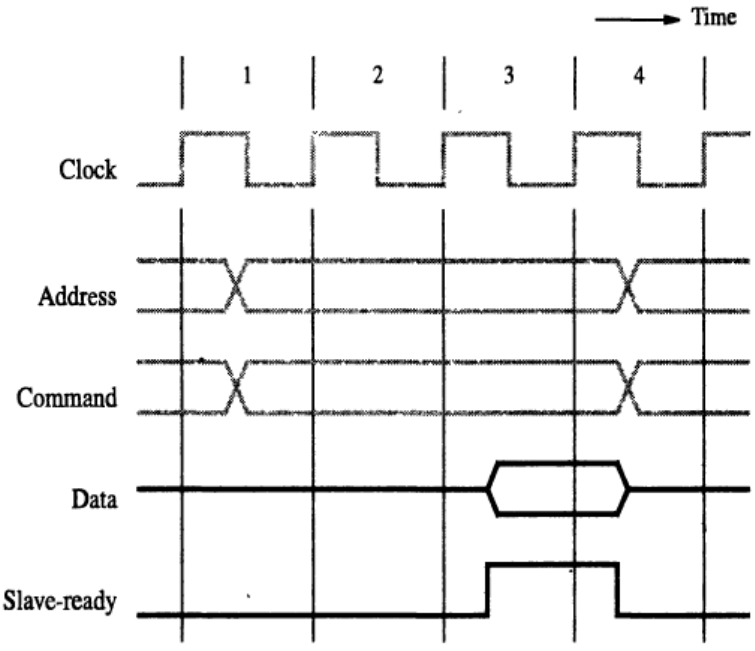


Figure 12.2: An input transfer using multiple clock cycles

ASYCHRONOUS BUS

An alternative scheme for controlling data transfers on the bus is based on the use of handshake between the master and the slave. The concept of a handshake is a generalization of the idea of the Slave-ready signal in Figure 12.2. The common clock is replaced by two timing control lines, Master-ready and Slave-ready. The first is asserted by the master to indicate that it is ready for a transaction, and the second is a response from the slave.

In principle, a data transfer controlled by a handshake protocol proceeds as follows. The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the Master-ready line. This causes all devices on the bus to decode the address. The selected slave performs the required operation and informs the processor it has done so by activating the Slave-ready line. The master waits for Slave-ready to become asserted before it removes its signals from the bus. In the case of a read operation, it also strobes the data into its input buffer.

I/O devices can be roughly categorized as storage, communications, user-interface, and other devices communicate with the computer via signals sent over wires or through the air. Devices connect with the computer via ***ports***, e.g. a serial or parallel port. A common set of wires connecting multiple devices is termed a ***bus.*** Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.

The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU). The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering). The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.

12.3 DETAILS OF I/O INTERFACE

An I/O interface is required whenever the I/O device is driven by the processor. The interface must have necessary logic to interpret the device address generated by the processor. The processor can communicate with an I/O device through the interface.

12.4 FUNCTIONS OF I/O INTERFACE

The following are the functions of I/O interface:

- I/O interface provides a storage buffer for one word of data.
- I/O interface contains status flag that can be accessed by the processor to determine whether the buffer is full or empty.
- I/O interface contains address-decoding circuitry to determine when it is being addressed by the processor.
- I/O interface enerates the appropriate timing signals required by the bus control scheme used
- I/O interface performs any format conversion that may be necessary to transfer data between the bus and the I/O device.

12.5 STANDARD I/O INTERFACES

The processor bus is the bus defined by the signals on the processor chip itself. Devices that require a very high speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a bridge that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices.

It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default,

when a particular design became commercially successful. For example, IBM developed a standard called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT. The popularity of that computer led to other manufacturers producing ISA -compatible interfaces for their 110 devices, thus making ISA into a de facto standard.

Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American Nationa1 Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.

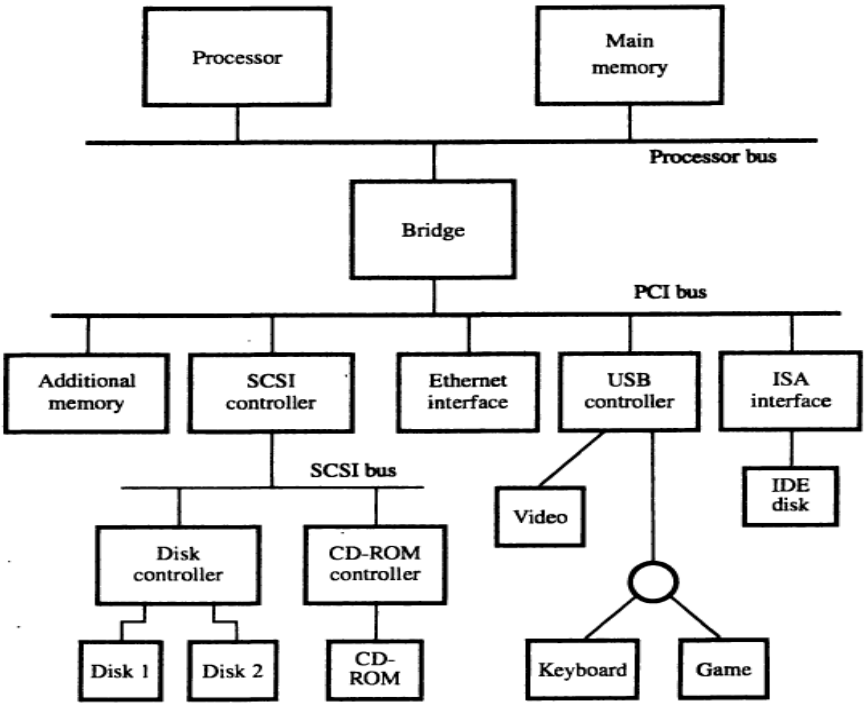


Figure 12.3: An example of a computer system using different interface standards

In this section, we present three widely used bus standards, PCI (Peripheral Component Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). The way these standards are used in a typical computer system is illustrated

in Figure 12.3. The PCI standard defines an expansion bus on the motherboard, SCSI and USB are used for connecting additional devices, both inside and outside the computer box. The SCSI bus is a high-speed parallel bus intended for devices such as disks and video displays. The USB bus uses serial transmission to suit the needs of equipment ranging from keyboards to game controls to internet connections. The figure shows an interface circuit that enables devices compatible with the earlier ISA standard, such as the popular IDE (Integrated Device Electronics) disk, to be connected. It also shows a connection to an Ethernet. The Ethernet is a widely used local area network, providing a high-speed connection among computers in a building or a university campus.

A given computer may use more than one bus standard. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

12.5.1 PCI BUS

The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC AT computers became known as the ISA bus. Its extended 32-bit version is known as the ISA bus. Other buses developed in the eighties with similar capabilities are the Micro channel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992.

An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus.

Data Transfer

In today's computers, most memory transfers involve a burst of data rather than just one word. The reason is that modern processors include a cache memory. Data are transferred between the cache and the main memory in bursts of several words each. The words involved in such a transfer are stored at successive memory locations. When the processor (actually the cache controller) specifies an address and requests a read operation from the main memory, the memory responds by sending a sequence of data words starting at that address. Similarly, during a write operation, the processor sends a memory address followed by a sequence of data words, to be written in successive memory locations starting at that address. The PCI is designed primarily to support this mode of operation. A read or a write operation involving a single word is simply treated as a burst of length one.

The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self-explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space. However, the system designer may choose to use memory-mapped I/O even when a separate I/O address space is available. In fact, this is the approach recommended by the PCI standard for wider compatibility. The configuration space is intended to give the PCI its plug-and-play capability. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

Figure 12.3 shows the main memory of the computer connected directly to the processor bus. An alternative arrangement that is used often with the PCI bus is shown in Figure 12.4. The PCI Bridge provides a separate physical connection for the main memory. For electrical reasons, the bus may be further divided into segments connected via bridges. However, regardless of which bus segment a device is connected to, it may still be mapped into the processor's memory address space.

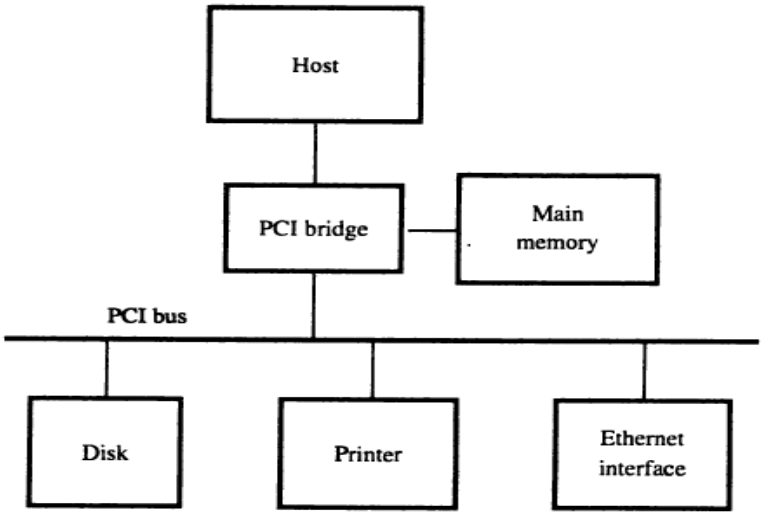


Figure 12.4: Use of a PCI bus in a computer system

Device Configuration

When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it. A typical device interface card for an ISA bus, for example, has a number of jumpers or switches that have to be set by the user to select certain options. Once the device is connected, the software needs to know the address of the device. It may also need to know relevant device characteristics, such as the speed of the transmission link, whether parity bits are used, and so on.

The PCI simplifies this process by incorporating in each I/O device interface a small configuration ROM memory that stores information about that device. The configuration ROMs of all devices are accessible in the configuration address space. The PCI initialization software reads these ROMs whenever the system is powered up or reset. In each case, it determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn about various device options and characteristics.

12.5.2 SCSI BUS:

The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131. In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.

The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time. There are also several options for the electrical signaling scheme used. The bus may use single-ended transmission (SE), where each signal uses one wire, with a common ground return for all signals. In another option, differential signaling is used, where a separate return wire is provided for each signal. In this case, two voltage levels are possible. Earlier versions use 5 V (TTL levels) and are known as High Voltage Differential (HVD). More recently, a 3.3 V version has been introduced and is known as Low Voltage Differential (LVD).

Because of these various options, the SCSI connector may have 50, 68, or 80 pins. The maximum transfer rate in commercial devices that are currently available varies from 5 megabytes/s to 160 megabytes/s. The most recent version of the standard is intended to support transfer rates up to 320 megabytes/s, and 640 megabytes/s is anticipated a little later. The maximum transfer rate on a given bus is often a function of the length of the cable and the number of devices connected, with higher rates for a shorter cable and fewer devices. To achieve the top data transfer rate, the bus length is typically limited to 1.6 m for SE signaling and 12 m for LVD signaling. However, manufacturers often provide special bus expanders to connect devices that are farther away. The maximum capacity of the bus is 8 devices for a narrow bus and 16 devices for a wide bus. Devices connected to the SCSI bus are not part of the address space of the processor in the same

way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller, as shown in Figure 12.3. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa. A packet may contain a block of data, commands from the processor to the device, or status information about the device.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory. Data are stored on a disk in blocks called sectors, where each sector may contain several hundred bytes. These data may not necessarily be stored in contiguous sectors. Some sectors may already contain previously stored data; others may be defective and must be skipped. Hence, a read or write request may result in accessing several disk sectors that are not necessarily contiguous. Because of the constraints of the mechanical motion of the disk, there is a long delay, in the order of several milliseconds, before reaching the first sector to or from which data are to be transferred. Then, a burst of data is transferred at high speed. Another delay may ensue, followed by a burst of data. A single read or write request may involve several such bursts. The SCSI protocol is designed to facilitate this mode of operation.

A controller connected to a SCSI bus is one of two types - an initiator or a target. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. Clearly, the controller on the processor side must be able to operate as an initiator. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target.

Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. While a particular connection is suspended, other devices can use the bus to transfer information. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and after winning arbitration, selects the controller it wants to communicate with and hands control of the

bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator.

Let us examine a complete disk read operation as an example. In this discussion, even though we refer to the initiator controller as taking certain actions, it should be clear that it performs these actions after receiving appropriate commands from the processor. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

- The SCSI controller, acting as an initiator, contends for control of the bus.
- When the 'initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
- The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
- The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
- The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
- The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
- The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator, as before. At the end of this transfer, the logical connection between the two controllers is terminated.

- ❖ As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
- ❖ The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. In this context, a "higher level" means that the messages refer to operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of operation of the particular device involved in a data transfer. In the preceding example, the processor need not be involved in the disk seek operations.

The SCSI bus standard defines a wide range of control messages that can be exchanged between the controllers to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

Bus Signal

We now describe the operation of the SCSI bus from the hardware point of view. The bus signals are summarized in Table 12.1. For simplicity we show the signals for a narrow bus (8 data lines). Note that all signal names are preceded by a minus sign. This indicates that the signals are active, or that a data line is equal to 1, when they are in the low-voltage state. The bus has no address lines. Instead, the data lines are used to identify the bus controllers involved during the selection or reselection process and during bus arbitration. For a narrow bus, there are eight possible controllers, numbered o through 7, and each i, associated with the data line that has the same number. A wide bus accommodates up to 16 controllers. A controller places its own address or the address of another controller on the bus by activating the corresponding data line. Thus, it is possible to have more than one address on the bus at the same time, as in the arbitration process we describe next. Once a connection is established between two controllers, there is no further need for addressing, and the data lines are used to carry data.

Table 12.1: The SCSI bus signals

Category	Name	Function
Data	–DB(0) to –DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	–DB(P)	Parity bit for the data bus
Phase	–BSY	Busy: Asserted when the bus is not free
	–SEL	Selection: Asserted during selection and reselection
Information type	–C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	–MSG	Message: indicates that the information being transferred is a message
Handshake	–REQ	Request: Asserted by a target to request a data transfer cycle
	–ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operation
Direction of transfer	–I/O	Input/Output: Asserted to indicate an input operation (relative to the initiator)
Other	–ATN	Attention: Asserted by an initiator when it wishes to send a message to a target
	–RST	Reset: Causes all device controls to disconnect from the bus and assume their start-up state

Check your progress:

1. Expand i) PCI Bus ii) SCSI Bus
2. How is data transfer performed in PCI Bus?
3. What is a bridge?

12.6 SUMMARY

In this unit, we discussed about I/O hardware. We also explained the popular interconnection standards are described namely the PCI, SCSI. They represent different approaches that meet the needs of various devices and reflect the increasing importance of plug-and-play features which increase user convenience.

12.7 KEYWORDS

I / O Interfaces,
PCI Bus
SCSI Bus

Answers to check your progress

12.8 ANSWER TO CHECK YOUR PROGRESS

1. 12.5
2. 12.5.1
3. 12.5

12.9 UNIT-END EXERCISES AND ANSWERS

1. Discuss the Bus signal of SCSI with a neat diagram.
2. Explain in detail about the I/O hardware and I/O interface.

Answer: SEE

1. 12.5.2
2. 12.4 and 12.5

12.10 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.
Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI. 1992

UNIT – 13: BASIC CONCEPTS

Structure

13.0 OBJECTIVES

After studying this module, you will be able to

- Discuss the organization of semiconductor memories
- Explain the various memory addressing schemes
- Discuss the connection between memory and CPU, the processor
- Explain the different types of memories

13.1 INTRODUCTION

A Memory is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. Programs and data they operate on are held in the main memory of the computer during execution. So here we discuss how this vital part of the computer operates.

A random access memory (RAM) differs from a read only memory (ROM) in that a RAM can transfer the stored information out (read) and is also capable of receiving new information in for storage (write).

13.2 SOME CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme.

Address	Memory Locations
16 Bit	2^{16} = 64 K
32 Bit	2^{32} = 4G (Giga)
40 Bit	2^{40} = IT (Tera)

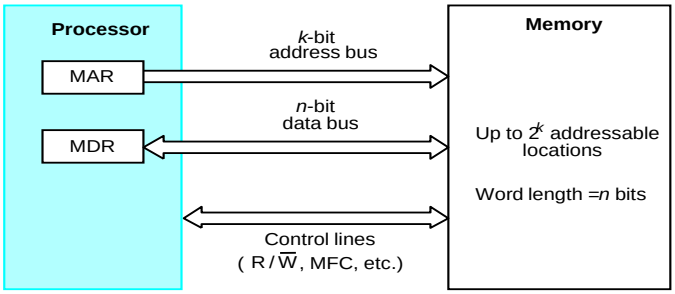


Figure 13.1: Connection of Memory to Processor

If MAR is k bits long and MDR is n bits long, then the memory may contain upto2^K addressable locations and the n-bits of data are transferred between the memory and processor. This transfer takes place over the processor bus.

The processor bus has,

- Address Line
- Data Line
- Control Line (R/W, MFC – Memory Function Completed)

The control line is used for coordinating data transfer. The processor reads the data from the memory by loading the address of the required memory location into MAR and setting the R/W line to 1. The memory responds by placing the data from the addressed location onto the data lines and confirms this action by asserting MFC signal.

Upon receipt of MFC signal, the processor loads the data onto the data lines into MDR register. The processor writes the data into the memory location by loading the address of this location into MAR and loading the data into MDR sets the R/W line to 0.

Memory Access Time →It is the time that elapses between the initiation of an operation and the completion of that operation.

Memory Cycle Time → It is the minimum time delay that is required between the initiations of the two successive memory operations.

13.3 MEMORY ADDRESSING

Let us consider the logical structure of a computer's random access memory (RAM). The generic term for the smallest unit of memory that the CPU can read or write is cell. In most modern computers, the size of a cell is 8 bits (1 byte). Hardware accessible units of memory larger than one cell are called words. Currently (1998) the most common word sizes are 32 bits (4 bytes) and 64 bits (8 bytes). Every memory cell has a unique integer address. The CPU accesses a cell by giving its address. Addresses of logically

adjacent cells differ by 1, the address space of a processor is the range of possible integer addresses, typically (0: $2^n - 1$)

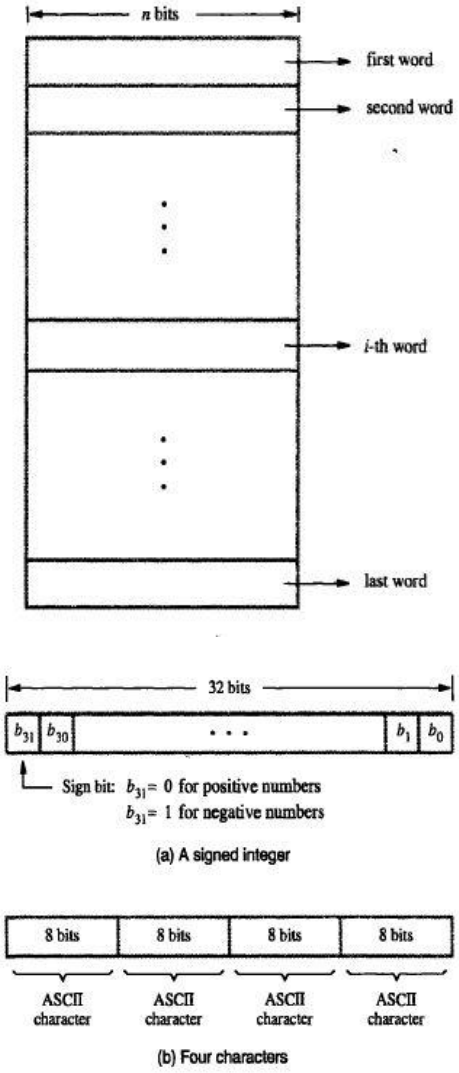


Figure 13.3: memory address

13.4 CPU MAIN-MEMORY CONNECTION

This topic will look at how data is transferred between the CPU and computer memory. We will do this by first looking at the technical detail of how bits are read and written between registers in the CPU and main memory. Once these details are understood we then look at actual instructions for loading and storing data in memory.

This is done by examining the SPASM instructions set. Data is transferred between registers in the CPU and memory cells. In this topic we will look at the different registers that are implemented in most CPUs. We will see that these are made up of registers that are accessible to the program and registers that control the operation of the CPU but are not directly accessible. Some of these registers are closely connected to the computer's internal buses and are used to read the buses and to change the status of the buses. We will also see how memory interprets the buses to read and write to memory cells.

After studying the details of data transfer we will examine an instruction set to see the effect of different interpretations of the data in the CPU. To do this we will learn about the SPASM computer and assembler language. SPASM is not a real computer. It is a simplified simulated computer designed as a learning tool. It takes instructions and ideas from several real computer systems. In this topic we only examine the instructions for accessing memory.

Accessing memory from the CPU

Before we look at how the CPU accesses memory we must expand our view of microcomputer architecture. We will first look at buses and then look at some of the registers internal to the CPU. We will then examine in more detail the interaction between the CPU and memory.

Buses

The buses that connect the three main parts of a computer system are called the control bus, the data bus and the address bus.

1. **Control bus** →is used to pass signals (0 and 1 bits) between the three components. For example, if the CPU wishes to read the contents of memory rather than write to memory it signals this on the control bus.
2. **Address bus**→ is used to address memory or I/O modules. Memory addresses or I/O module port addresses are placed on this bus.

3. **Data bus**→ is used to transfer data. After the control bus signals a read or write operation and the address is placed on the address bus the computer component concerned places the data on the data bus so the destination can read it

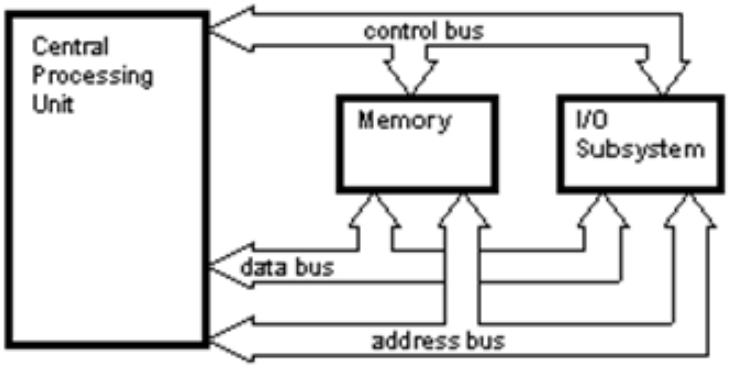


Figure 13.2: The computer's buses

As an example of the use of these buses, we can list the steps involved in the CPU reading data from an address in memory. These steps are:

- The CPU places the memory address on the address bus
- The CPU requests a memory read operation on the control bus
- The memory recognizes the memory read operation and examines the address on the address bus
- Memory moves the data from cells at the address on the address bus to the data bus
- The CPU reads the data off the data bus.

13.5 MEMORY ACCESS TYPE

There are two types of memory access.

- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)

Uniform Memory Access (UMA) is a shared memory architecture used in parallel computers.

All the processors in the UMA model share the physical memory uniformly. In UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.

Uniform Memory Access computer architectures are often contrasted with Non-Uniform Memory Access (NUMA) architectures.

In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion; The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

Non-Uniform Memory Access (**NUMA**) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. They were developed commercially during the 1990s by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Honeywell Information Systems Italy (HISI) (later Group Bull), Silicon Graphics (later Silicon Graphics International), Sequent Computer Systems (later IBM), Data General (later EMC), and Digital (later Compaq, now HP). Techniques developed by these companies later featured in a variety of Unix like operating systems, and to an extent in Windows NT.

The first commercial implementation of a NUMA-based UNIX system was the Symmetrical Multi Processing XPS-100 family of servers, designed by Dan Gielan of VAST Corporation for Honeywell Information Systems Italy. The tremendous success of the architecture made HISI the top UNIX vendor in Europe.

13.6 MEMORY CYCLE TIME

Memory access time is how long it takes for a character in memory to be transferred to or from the CPU. In a PC or Mac, fast RAM chips have an access time of 70 nanoseconds (ns) or less. SDRAM chips have a burst mode that obtains the second and subsequent characters in 10 ns

or less. Disk access time is an average of the time between initiating a request and obtaining the first data character. It includes the command processing, the average seek time (moving the read/write head to the required track) and the average latency (rotation of disk to the required sector). This specification must be given as an average, because seek times and latency can vary depending on the current position of the head and platter. Fast hard disks have access times of 10 milliseconds (ms) or less. This is a common speed measurement, but overall disk performance is significantly influenced by channel speed (transfer rate), interleaving and caching.

13.7 Random Access Memory (RAM)

In RAM, if any location that can be accessed for a Read/Write operation in fixed amount of time, it is independent of the locations address. **RAM** is a form of computer data storage. A random access device allows stored data to be accessed in very nearly the same amount of time for any storage location, so data can be accessed quickly in any random order. In contrast, other data storage media such as shared disks, CDs, DVDs and magnetic tape read and write data only in a predetermined order, consecutively, because of mechanical design limitations. Therefore the time to access a given data location varies significantly depending on its physical location.

Today, random access memory takes the form of integrated circuits. Strictly speaking, modern types of DRAM are not random access, as data is read in bursts, although the name DRAM / RAM has stuck. However, many types of SRAM, ROM, OTP, and NOR flash are still random access even in a strict sense. RAM is often associated with volatile types of memory (such as DRAM memory modules), where its stored information is lost if the power is removed. Many other types of non-volatile memory are

RAM as well, including most types of ROM and a type of flash memory called *NOR-Flash*. The first RAM modules to come into the market were created in 1951 and were sold until the late 1960s and early 1970s.

Types of RAM

The two main forms of modern RAM are static RAM (SRAM) and dynamic RAM (DRAM). In static RAM, a bit of data is stored using the state of a flip-flop. This form of RAM is more expensive to produce, but is generally faster and requires less power than DRAM and, in modern computers, is often used as cache memory for the CPU. DRAM stores a bit of data using a transistor and capacitor pair, which together comprise a memory cell. The capacitor holds a high or low charge (1 or 0, respectively), and the transistor acts as a switch that lets the control circuitry on the chip read the capacitor's state of charge or change it. As this form of memory is less expensive to produce than static RAM, it is the predominant form of computer memory used in modern computers.

Both static and dynamic RAM's are considered *volatile*, as their state is lost or reset when power is removed from the system. By contrast, Read only memory (ROM) stores data by permanently enabling or disabling selected transistors, such that the memory cannot be altered. Writeable variants of ROM (such as EEPROM and flash memory) share properties of both ROM and RAM, enabling data to persist without power and to be updated without requiring special equipment. These persistent forms of semiconductor ROM include USB flash drives, memory cards for cameras and portable devices, etc. As of 2007, NAND flash has begun to replace older forms of persistent storage, such as magnetic disks and tapes, while NOR flash is being used in place of ROM in net-books and rugged computers, since it is capable of true random access, allowing direct code execution.

ECC memory (which can be either SRAM or DRAM) includes special circuitry to detect and/or correct random faults (memory errors) in the stored data, using parity bits or error correction code.

In general, the term *RAM* refers solely to solid state memory devices (either DRAM or SRAM), and more specifically the main memory in most computers. In optical storage,

the term DVD-RAM is somewhat of a misnomer since, unlike CD-RW or DVD-RW it does not require to be erased before reuse. Nevertheless a DVD-RAM behaves much like a hard disc drive if somewhat slower.

13.8 Cache Memory

It is a small, fast memory that is inserted between the larger slower main memory and the processor. It holds the currently active segments of a program and their data. The *cache* is a small amount of high speed memory, usually with a memory cycle time comparable to the time required by the CPU to fetch one instruction. The cache is usually filled from main memory when instructions or data are fetched into the CPU. Often the main memory will supply a wider data word to the cache than the CPU requires, to fill the cache more rapidly. The amount of information which replaces at one time in the cache is called the *line size* for the cache. This is normally the width of the data bus between the cache memory and the main memory. A wide line size for the cache means that several instruction or data words are loaded into the cache at one time, providing a kind of prefetching for instructions or data. Since the cache is small, the effectiveness of the cache relies on the following properties of most programs:

Spatial locality -- most programs are highly sequential; the next instruction usually comes from the next memory location.

Data is usually structured, and data in these structures normally are stored in contiguous memory locations. Short loops are a common program structure, especially for the innermost sets of nested loops. This means that the same small set of instructions is used over and over. Generally, several operations are performed on the same data values, or variables.

When a cache is used, there must be some way in which the memory controller determines whether the value currently being addressed in memory is available from the cache. There are several ways that this can be accomplished. One possibility is to store both the address and the value from main memory in the cache, with the address stored in a type of memory called *associative memory* or, more descriptively, *content addressable memory*.

An associative memory, or content addressable memory, has the property that when a value is presented to the memory, the *address* of the value is returned if the value is stored in the memory, otherwise an indication that the value is not in the associative memory is returned. *All* of the comparisons are done simultaneously, so the search is performed very quickly. This type of memory is very expensive, because each memory location must have both a comparator and a storage element. A cache memory can be implemented with a block of associative memory, together with a block of ``ordinary'' memory. The associative memory would hold the *address* of the data stored in the cache, and the ordinary memory would contain the data at that address.

13.9 Virtual memory

The address generated by the processor does not directly specify the physical locations in the memory. The address generated by the processor is referred to as a virtual / logical address. The virtual address space is mapped onto the physical memory where data are actually stored. The mapping function is implemented by a special memory control circuit is often called the memory management unit. Only the active portion of the address space is mapped into locations in the physical memory. The remaining virtual addresses are mapped onto the bulk storage devices used, which are usually magnetic disk.

As the active portion of the virtual address space changes during program execution, the memory management unit changes the mapping function and transfers the data between disk and memory. Thus, during every memory cycle, an address processing mechanism determines whether the addressed in function is in the physical memory unit. If it is, then the proper word is accessed and execution proceeds. If it is not, a page of words containing the desired word is transferred from disk to memory. This page displaces some page in the memory that is currently inactive.

Check your progress:

1. Expand RAM and ROM.
2. What are the different types of lines processor bus has?
3. Define Memory Access Time and Memory Cycle Time.

13.10 SUMMARY

In this unit, we prescribed the most important technological and organizational details of memory, memory speed and how apparent speed of main memory can be increased by means of caches. We also discussed about virtual memory concepts which increases the apparent size in main memory.

13.11 KEYWORDS

Memory

Cache Memory

RAM

DRAM

ROM

SRAM

Virtual memory

13.12 ANSWER TO CHECK YOUR PROGRESS

1. 13.1
2. 13.2
3. 13.2

13.13 UNIT-END EXERCISES AND ANSWERS

1. Explain the basic concepts of memory.
2. Explain different memory access types.
3. Write short notes on the following.
    a) RAM
    b) Virtual Memory
    c) Cache Memory

Answer: SEE

1. 13.2
2. 13.5
3. 13.7, 13,8, 13.9

13.14 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.

Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI. 1992

UNIT – 14: SEMICONDUCTOR RAM MEMORIES

Structure

14.0 OBJECTIVES

After studying this module, you will be able to

- Discuss the organization of semiconductor memories.
- Explain the different types of memories.
- Memory Hierarchy

14.1 INTRODUCTION

The basic technology for implementing main memories uses semiconductor integrated circuits. Semiconductor memories are available in a wide range of speeds. Due to rapid advances in VLSI technology the cost of semiconductor memories has dropped

dramatically. In this unit we will present the main characteristics of semiconductor memories.

14.2 INTERNAL ORGANIZATION OF MEMORY CHIPS:

Memory cells are usually organized in the form of array, in which each cell is capable of storing one bit of information. Each row of cells constitutes a memory word and all cells of a row are connected to a common line called as word line. The cells in each column are connected to Sense / Write circuit by two bit lines. Figure 14.1 shows the possible arrangements of memory cells.

The Sense / Write circuits are connected to data input or output lines of the chip. During a write operation, the sense / write circuit receives input information and stores it in the cells of the selected word. The data input and data output of each sense / write circuits are connected to a single bidirectional data line that can be connected to a data bus of the cpu.

R / W → specifies the required operation.

CS → Chip Select input selects a given chip in the multi-chip memory system

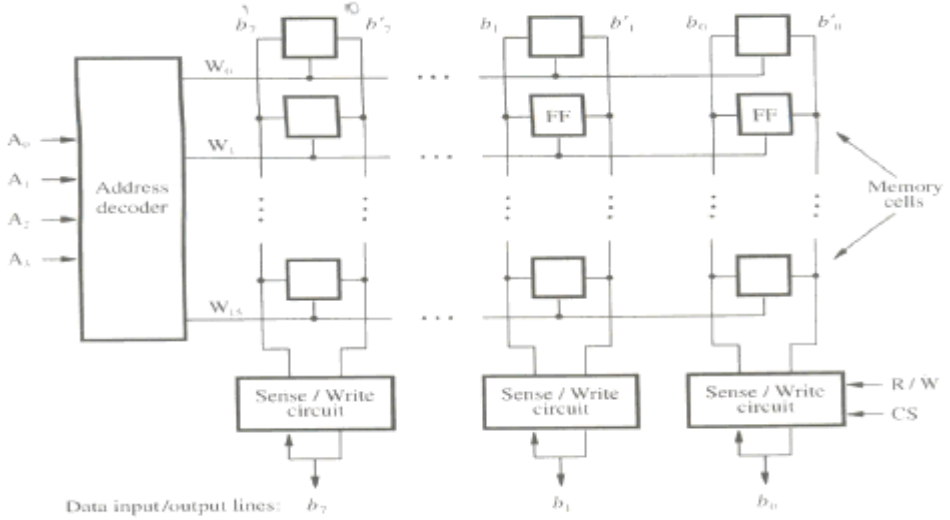


Figure 14.1: Organization of bit cells in a memory chip

14.3 SEMI CONDUCTOR RAM MEMORIES:

Semi-Conductor memories are available is a wide range of speeds. Their cycle time ranges from 100ns to 10ns. When first introduced in the late 1960s, they were much more expensive. But now they are very cheap, and used almost exclusively in implementing main memories.

14.4 STATIC MEMORIES:

Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memory.

Static random-access memory (**SRAM**) is a type of semiconductor memory that uses bistable latching circuitry to store each bit. The term *static* differentiates it from *dynamic* RAM (DRAM) which must be periodically refreshed. SRAM exhibits data remanence, but is still *volatile* in the conventional sense that data is eventually lost when the memory is not powered. Figure 14.2 shows the implementation of static RAM.

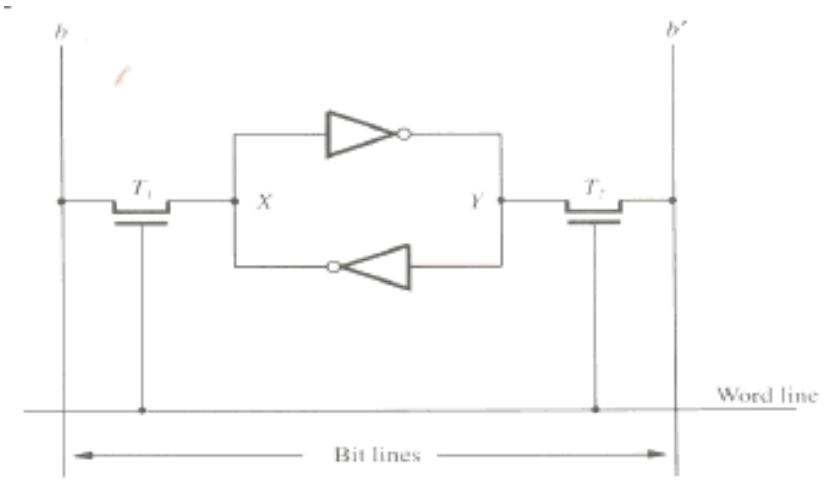


Figure 14.2: Static RAM cell

Two inverters are cross connected to form a latch. The latch is connected to two bit lines by transistors T_1 and T_2. These transistors act as switches that can be opened / closed under the control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state.

Read Operation:

In order to read the state of the SRAM cell, the word line is activated to close switches T_1 and T_2. If the cell is in state 1, the signal on bit line b is high and the signal on the bit line b is low. Thus b and b' are complement of each other. Sense / write circuit at the end of the bit line monitors the state of b and b' and set the output according.

Write Operation:

The state of the cell is set by placing the appropriate value on bit line b and its complement on b' and then activating the word line. This forces the cell into the corresponding state. The required signal on the bit lines are generated by Sense / Write circuit.

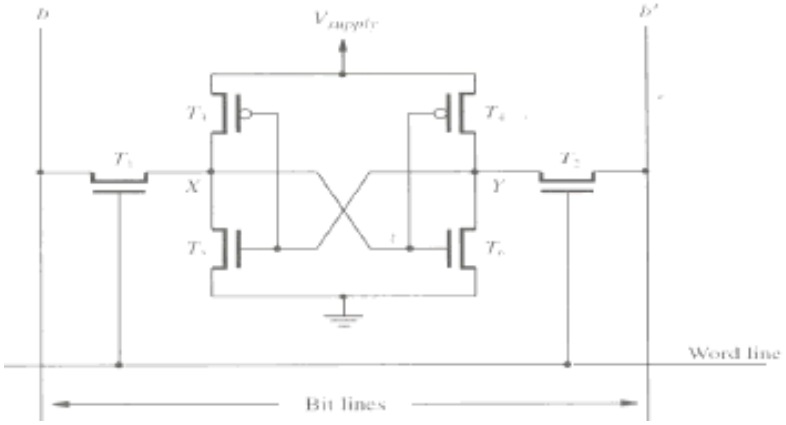


Figure 14.3: CMOS cell (Complementary Metal oxide Semi-Conductor):

Transistor pairs (T_3, T_5) and (T_4, T_6) form the inverters in the latch. In state 1, the voltage at point X is high by having T_5, T_6 on and T_4, T_5 are OFF. Thus T_1 and T_2 returned ON (Closed), bit line b and b' will have high and low signals respectively. The CMOS requires 5V (in older version) or 3.3.V (in new version) of power supply voltage. The

continuous power is needed for the cell to retain its state. A CMOS cell realization is shown in Figure 14.3.

Merit:

- It has low power consumption because the current flows in the cell only when the cell is being accessed.
- Static RAMs can be accessed quickly. It access time is few nanoseconds.

Demerit:

- SRAMs are said to be volatile memories because their contents are lost when the power is interrupted.

Asynchronous DRAMS:

Less expensive RAM‟s can be implemented if simpler cells are used. Such cells cannot retain their state indefinitely. Hence they are called Dynamic RAM's (DRAM). The information stored in a dynamic memory cell in the form of a charge on a capacitor and this charge can be maintained only for a few milliseconds. The contents must be periodically refreshed by restoring this capacitor charge to its full value.

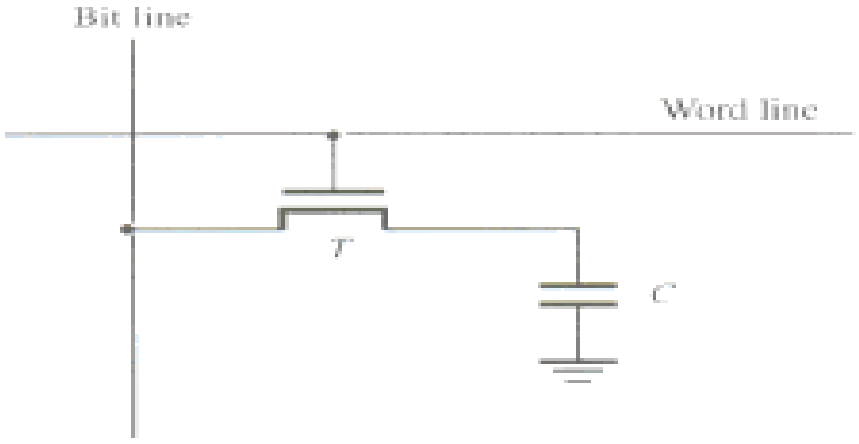


Figure 14.4: A single transistor dynamic Memory cell

In order to store information in the cell, the transistor T is turned on and the appropriate voltage is applied to the bit line, which charges the capacitor. After the transistor is turned off, the capacitor begins to discharge which is caused by the capacitor's own leakage resistance. Hence the information stored in the cell can be retrieved correctly before the threshold value of the capacitor drops down, as shown in Figure 14.4.

During a read operation, the transistor is turned on and a sense amplifier connected to the bit line detects whether the charge on the capacitor is above the threshold value. A 16-megabit DRAM chip configured as 2M x 8, is shown in Figure 14.5.

If charge on capacitor > threshold value →Bit line will have logic value **1**.
If charge on capacitor < threshold value → Bit line will set to logic value **0**.

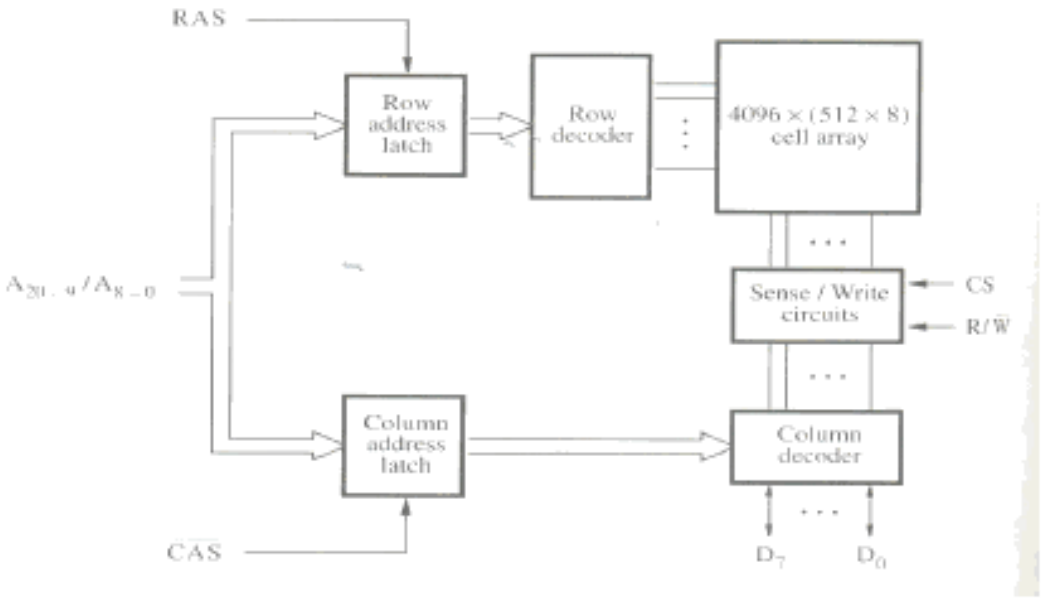


Figure 14.5: Internal organization of a 2M X 8 dynamic Memory chip.

DESCRIPTION:

The 4 bit cells in each row are divided into 512 groups of 8. 21 bit address is needed to access a byte in the memory (12 bit to select a row, and 9 bits specify the group of 8 bits in the selected row).

A (0-8) → Row address of a byte.

A (9-20) → Column address of a byte.

During Read/ Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on Row Address Strobe (RAS) input of the chip. When a Read operation is initiated, all cells on the selected row are read and refreshed. Shortly after the row address is loaded, the column address is applied to the address pins and loaded into Column Address Strobe (CAS). The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits is selected. R/W =1(read operation).

The output values of the selected circuits are transferred to the data lines D0 - D7. R/W=0 (write operation). The information on D0 - D7 is transferred to the selected circuits.

RAS and CAS are active low so that they cause the latching of address when they change from high to low. This is because they are indicated by RAS and CAS. To ensure that the contents of a DRAM's are maintained, each row of cells must be accessed periodically. Refresh operation usually perform this function automatically. A specialized memory controller circuit provides the necessary control signals RAS and CAS that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are referred to as Asynchronous DRAM's.

Fast Page Mode:

Transferring the bytes in sequential order is achieved by applying the consecutive sequence of column address under the control of successive CAS signals. This scheme allows transferring a block of data at a faster rate. The block of transfer capability is called as Fast Page Mode.

Synchronous DRAM:

Here the operations are directly synchronized with clock signal. The address and data connections are buffered by means of registers. The output of each sense amplifier is

connected to a latch. A Read operation causes the contents of all cells in the selected row to be loaded in these latches. The Figure 14.6 shows the structure of SDRAM.

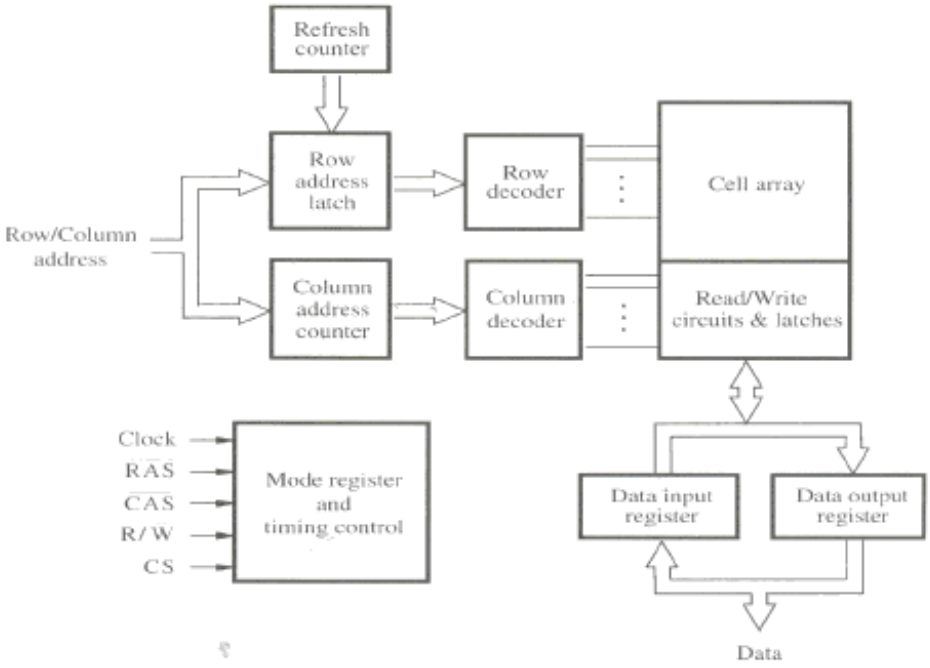


Figure 14.6: Synchronous DRAM

Data held in the latches that correspond to the selected columns are transferred into the data output register, thus becoming available on the data output pins.

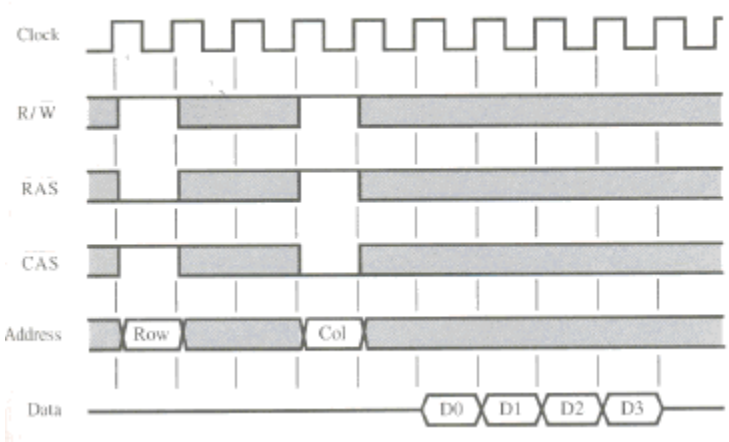


Figure 14.7: Timing Diagram Burst Read of Length 4 in an SDRAM

First, the row address is latched under control of RAS signal. The memory typically takes 2 or 3 clock cycles to activate the selected row. Then the column address is latched under the control of CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next 3 sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles. A timing diagram for a typical burst of read of length 4 is shown in Figure 14.7.

Latency and Bandwidth:

A good indication of performance is given by two parameters. They are,

- Latency
- Bandwidth

Latency refers to the amount of time it takes to transfer a word of data to or from the memory. For a transfer of single word, the latency provides the complete indication of memory performance. For a block transfer, the latency denotes the time it takes to transfer the first word of data.

Bandwidth is defined as the number of bits or bytes that can be transferred in one second. Bandwidth mainly depends upon the speed of access to the stored data and on the number of bits that can be accessed in parallel.

Double Data Rate SDRAM (DDR-SDRAM):

The standard SDRAM performs all actions on the rising edge of the clock signal. The double data rate SDRAM transfer data on both the edges (loading edge, trailing edge). The Bandwidth of DDR-SDRAM is doubled for long burst transfer. To make it possible to access the data at high rate, the cell array is organized into two banks. Each bank can be accessed separately. Consecutive words of a given block are stored in different banks. Such interleaving of words allows simultaneous access to two words that are transferred on successive edge of the clock.

14.5 Dynamic Memory System:

The physical implementation is done in the form of Memory Modules. If a large memory is built by placing DRAM chips directly on the main system printed circuit board that contains the processor, often referred to as Motherboard; it will occupy large amount of space on the board. These packaging consideration have led to the development of larger memory units known as SIMM's and DIMM's

SIMM-Single Inline memory Module
DIMM-Dual Inline memory Module

SIMM and DIMM consist of several memory chips on a separate small board that plugs vertically into single socket on the motherboard.

MEMORY SYSTEM CONSIDERATION:

To reduce the number of pins, the dynamic memory chips use multiplexed address inputs. The address is divided into two parts. They are,

High Order Address Bit (Select a row in cell array and it is provided first and latched into memory chips under the control of RAS signal).

Low Order Address Bit (Selects a column and they are provided on same address pins and latched using CAS signals).

The Multiplexing of address bit is usually done by Memory Controller Circuit, as shown in Figure 14.8.

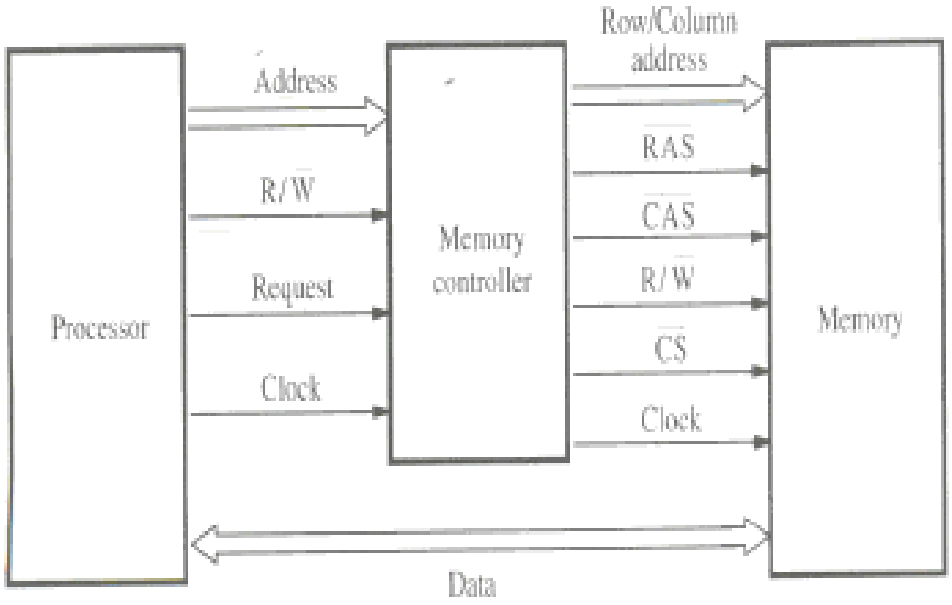


Figure 14.8: Use of Memory Controller

The Controller accepts a complete address and R/W signal from the processor, under the control of a request signal which indicates that a memory access operation is needed. The Controller then forwards the row and column portions of the address to the memory and generates RAS and CAS signals. It also sends R/W and CS signals to the memory. The CS signal is usually active low, hence it is shown as CS.

Refresh Overhead:

All dynamic memories have to be refreshed. In DRAM, the period for refreshing all rows is 16ms whereas 64ms in SDRAM.

Example: Given a cell array of 8K (8192).

Clock cycle=4

Clock Rate=133MHZ

No of cycles to refresh all rows = 8192*4 =32,768

Time needed to refresh all rows = 32768/133*106 =246*10-6 sec =0.246sec

Refresh Overhead = 0.246/64

Refresh Overhead = 0.003

Rambus Memory:

The usage of wide bus is expensive. Rambus developed the implementation of narrow bus. Rambus technology is a fast signaling method used to transfer information between chips. Instead of using signals that have voltage levels of either 0 or Vsupply to represent the logical values, the signals consist of much smaller voltage swings around a reference voltage Vref. The reference Voltage is about 2V and the two logical values are represented by 0.3V swings above and below Vref.

This type of signaling is generally is known as Differential Signaling. Rambus provides a complete specification for the design of communication links (Special Interface circuits) called as Rambus Channel. Rambus memory has a clock frequency of 400MHZ. The data are transmitted on both the edges of the clock so that the effective data transfer rate is 800MHZ.

The circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as Rambus DRAMs (RDRAM).
Rambus channel has,

- 9 Data lines (1-8 Transfer the data, 9th line → Parity checking).
- Control line
- Power line

A two channel Rambus has 18 data lines which have no separate address lines. It is also called as Direct RDRAM's. Communication between processor or some other device that can serves as a master and RDRAM modules are served as slaves, is carried out by means of packets transmitted on the data lines.

There are 3 types of packets. They are,

- Request
- Acknowledge
- Data

14.6 READ ONLY MEMORY (ROM):

Both SRAM and DRAM chips are volatile, which means that they lose the stored information if power is turned off. Many applications require Non-volatile memory (which retains the stored information if power is turned off).

E.g.: Operating System software has to be loaded from disk to memory which requires the program that boots the Operating System. i.e., it requires non-volatile memory. Non-volatile memory is used in embedded system. Since the normal operation involves only reading of stored data, a memory of this type is called ROM.

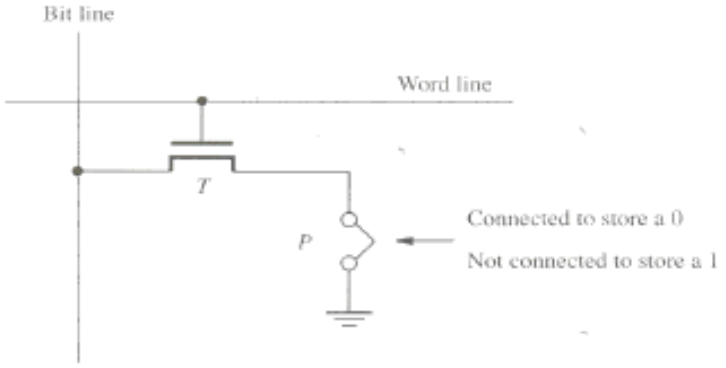


Figure 14.9: ROM cell

At Logic value '0' → Transistor (T) is connected to the ground point (P). Transistor switch is closed and voltage on bit line nearly drops to zero.
At Logic value '1' → Transistor switch is open. The bit line remains at high voltage. To read the state of the cell, the word line is activated. A Sense circuit at the end of the bit line generates the proper output value.

Different types of non-volatile memory are:

- PROM
- EPROM
- EEPROM
- Flash Memory

PROM: Programmable ROM:

PROM allows the data to be loaded by the user. Programmability is achieved by inserting a fuse at point P in a ROM cell. Before it is programmed, the memory contains all 0's. The user can insert 1's at the required location by burning out the fuse at these locations using high current pulse. This process is irreversible.

Merit:

- It provides flexibility.
- It is faster.
- It is less expensive because they can be programmed directly by the user.

EPROM - Erasable reprogrammable ROM:

EPROM allows the stored data to be erased and new data to be loaded. In an EPROM cell, a connection to ground is always made at 'P' and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off. This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside.

Erasure requires dissipating the charges trapped in the transistor of memory cells. This can be done by exposing the chip to ultraviolet light, so that EPROM chips are mounted in packages that have transparent windows.

Merits:

- It provides flexibility during the development phase of digital system.
- It is capable of retaining the stored information for a long time.

Demerits:

- The chip must be physically removed from the circuit for reprogramming and its entire contents are erased by UV light.

EEPROM:-Electrically Erasable ROM:

EEPROM (also written **E^{2}PROM** and pronounced "e-e-prom," "double-e prom," "e-squared," or simply "e-prom") stands for **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration.

When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices. EEPROMs are realized as arrays of floating-gate transistors. EEPROM is user modifiable read only memory (ROM) that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage generated externally or internally in the case of modern EEPROMs. EPROM usually must be removed from the device for erasing and programming, whereas EEPROMs can be programmed and erased in circuit. Originally, EEPROMs were limited to single byte operations which made them slower, but modern EEPROMs allow multi-byte page operations. It also has a limited life - that is, the number of times it could be reprogrammed was limited to tens or hundreds of thousands of times. That limitation has been extended to a million write operations in modern EEPROMs. In an EEPROM that is frequently reprogrammed while the computer is in use, the life of the EEPROM can be an important design consideration. It is for this reason that EEPROMs were used for configuration information, rather than random access memory.

Merits:

It can be both programmed and erased electrically.

It allows the erasing of all cell contents selectively.

Demerits:

It requires different voltage for erasing, writing and reading the stored data.

FLASH MEMORY:

In EEPROM, it is possible to read and write the contents of a single cell. In Flash device, it is possible to read the contents of a single cell but it is only possible to write the entire contents of a block. Prior to writing, the previous contents of the block are erased.

E.g.: In MP3 player, the flash memory stores the data that represents sound.

Single flash chips cannot provide sufficient storage capacity for embedded system application. There are 2 methods for implementing larger memory modules consisting of number of chips. They are,

- Flash Cards
- Flash Drives.

Merits:

- Flash drives have greater density which leads to higher capacity and low cost per bit.
- It requires single power supply voltage and consumes less power in their operation.

Flash Cards:

One way of constructing larger module is to mount flash chips on a small card. Such flash card have standard interface. The card is simply plugged into a conveniently accessible slot. Its memory size is of 8, 32,64MB. E.g.: A minute of music can be stored in 1MB of memory. Hence 64MB flash cards can store an hour of music.

Flash Drives:

Larger flash memory module can be developed by replacing the hard disk drive. The flash drives are designed to fully emulate the hard disk. The flash drives are solid state electronic devices that have no movable parts.

Merits:

- They have shorter seek and access time which results in faster response.

- They have low power consumption which makes them attractive for battery driven application.
- They are insensitive to vibration.

Demerit:

- The capacity of flash drive (<1GB) is less than hard disk (>1GB).
- It leads to higher cost per bit.
- Flash memory will deteriorate after it has been written a number of times (typically at least 1 million times.)

SPEED, SIZE COST:

Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

Magnetic Disk:

A huge amount of cost effective storage can be provided by magnetic disk. The main memory can be built with DRAM which leaves SRA's to be used in smaller units where speed is of essence.

Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

14.7 MEMORY HIERARCHY

To this point in our study of systems, we have relied on a simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU. In our simple \model, the memory system is a linear array of bytes, and the CPU can access each memory location in a constant amount of time. While this is an effective model as far as it goes, it does not reflect the way that modern systems really work. In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. CPU registers hold the most frequently used data. Small, fast cache memories nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because well written programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fast storage near the top of the hierarchy. As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data your program needs are stored in a CPU register, then they can be accessed in zero cycles during the execution of the instruction. If stored in a cache, 1 to 30 cycles. If stored in main memory, 50 to 200 cycles. And if stored in disk tens of millions of cycles!. The entire computer memory can be realized as the hierarchy shown in the Figure 14.10.

Here, then, is a fundamental and enduring idea in computer systems: If you understand how the system moves data up and down the memory hierarchy, then you can write your application programs so that their data items are stored higher in the hierarchy, where the CPU can access them more quickly. This idea centers on a fundamental property of computer programs known as locality. Programs with good locality tend to access the same set of data items over and over again, or they tend to access sets of nearby data items. Programs with good locality tend to access more data items from the upper levels of the memory hierarchy than programs with poor locality, and thus run faster. For example, the running times of different matrix multiplication kernels that perform the

same number of arithmetic operations, but have different degrees of locality, can vary by a factor of 20!

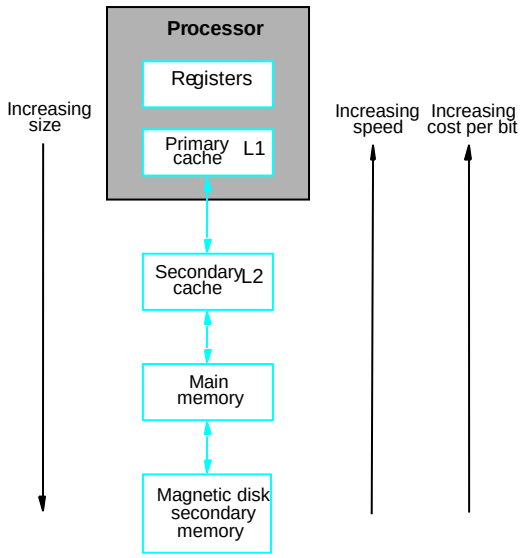


Figure 14.10: Memory Hierarchy

Types of Cache Memory:

The Cache memory is of 2 types. They are:

- Primary /Processor Cache (Level1 or L1 cache)
- Secondary Cache (Level2 or L2 cache)

Primary Cache → It is always located on the processor chip.

Secondary Cache → It is placed between the primary cache and the rest of the memory.

The main memory is implemented using the dynamic components (SIMM, RIMM, and DIMM). The access time for main memory is about 10 times longer than the access time for L1 cache.

Check your progress:

1. What is a semiconductor memory?
2. What is RAMBUS memory?
3. What are the different types of memory? Explain briefly the different operations performed on static memories.

14.8 SUMMARY

In this unit we discussed the common components and organizations used to implement the main memory. Then we examined memory speed and discussed how the apparent speed of the main memory can be increased by means of caches. We also discussed about the concept of ROM, it verities, their significance, architecture, uses and their limitations. We finally concluded with hierarchy of memory.

14.9 KEYWORDS

Memory
Memory chips
RAM, DRAM, SDRAM
RAMBUS
ROM, PROM, EPROM, EEPROM, Flash Memory

14.10 ANSWERS TO CHECK YOUR PROGRESS

1. 14.1
2. 14.5
3. 14.3..14.6

14.11 UNIT-END EXERCISES AND ANSWERS

1. Discuss the importance of memory in a computing system.
2. Expand RAM. Also explain semiconductor RAM memories.
3. With a neat diagram, explain the organization of a memory chip.

Answers: SEE

1 14.2

2 14.3

3 14.2

14.12 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.

Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI. 1992

UNIT – 15: CACHE MEMORIES

Structure

15.0 OBJECTIVES

After going through this module, you should be able to

- Understand the concept of Cache Memories
- Discuss their mapping functions,
- Replacement algorithms
- Performance considerations.

15.1 INTRODUCTION

The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are

cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory. In this unit, we discuss the concepts related to cache memory, the need for it, their architecture, their working.

15.2 CACHE MEMORY CONCEPT

Cache memory is an integral part of every system now. Cache memory is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. As the microprocessor processes data, it looks first in the cache memory and if it finds the data there (from a previous reading of data), it does not have to do the more time consuming reading of data from larger memory.

The effectiveness of cache mechanism is based on the property of Locality of reference.

Locality of Reference:

During some time period and remainder of the program is accessed relatively infrequently. It manifests itself in 2 ways. They are Temporal (The recently executed instruction are likely to be executed again very soon), Spatial (The instructions in close proximity to recently executed instruction are likely to be executed soon). If the active segment of the program is placed in cache memory, then the total execution time can be reduced significantly.

If the active segment of a program can be placed in a fast cache memory, then the total execution time can be reduced significantly. The operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. The term Block refers to the set of contiguous address locations of some size. The cache line is used to refer to the cache block.

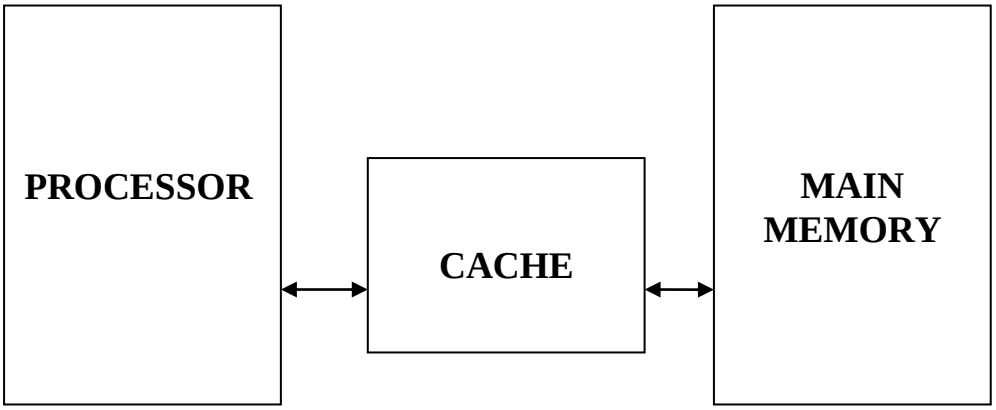


Figure 15.1: Use of Cache Memory

The Figure 15.1 shows arrangement of Cache between processor and main memory. The Cache memory stores a reasonable number of blocks at a given time but this number is small compared to the total number of blocks available in Main Memory. The correspondence between main memory block and the block in cache memory is specified by a mapping function. The Cache control hardware decides that which block should be removed to create space for the new block that contains the referenced word. The collection of rule for making this decision is called the replacement algorithm. The cache control circuit determines whether the requested word currently exists in the cache. If it exists, then Read/Write operation will take place on appropriate cache location. In this case Read/Write hit will occur. In a Read operation, the memory will not be involved.

The write operation proceeds in 2 ways. They are:

- Write-through protocol
- Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit. The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block. If the requested word currently does not exist in the cache during read operation, then read miss will occur. To overcome the read miss Load –through / early restart protocol is used.

Read Miss:

The block of words that contains the requested word is copied from the main memory into cache.

Load –through:

After the entire block is loaded into cache, the particular word requested is forwarded to the processor. If the requested word does exist in the cache during write operation, then Write Miss will occur. If Write through protocol is used, the information is written directly into main memory. If Write back protocol is used then blocks containing the addressed word is first brought into the cache and then the desired word in the cache is overwritten with the new information.

15.3 CACHE MEMORY DESIGN PARAMETERS

There are two cache design parameters that dramatically influence the cache performance: the block size and the cache associability. There are also many other implementation techniques both hardware and software that improve the cache performance but they are not discussed here.

The simplest way to reduce the miss rate is to increase the block size. However increasing the block size also increases the miss penalty (which is the time to load a block from main memory into cache) so there is a trade–off between the block size and miss penalty. We can increase the block size up to a level at which the miss rate is decreasing

but we also have to be sure that this benefit will not be exceeded by the increased miss penalty.

The second cache design parameter that reduces cache misses is the associability. There is an empirical result called the 2:1 rule of thumb which states that a direct mapped cache of size N has about the same miss rate as a 2 way set associative cache of size N/2. Unfortunately an increased associability will have a bigger hit time. More time will be taken to retrieve a block inside of an associative cache than in a direct mapped cache. To retrieve a block in an associative cache, the block must be searched inside of an entire set since there is more than one place where the block can be stored.

Based on the cause that determines a cache miss we can classify the cache misses as compulsory, capacity and conflict misses. This classification is called the 3C model. Compulsory misses are issued when a first access is done to a block that is not in the memory, so the block must be brought into cache. Increasing block size can reduce compulsory misses due to prefetching the other elements in the block. If the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Increasing the associability in general reduce the number of conflict misses and implicitly the runtime of the programs. However this is not true all the time. Minimizing the cache misses does not necessarily minimize the runtime. For example, there can be fewer cache misses with more memory accesses.

15.4 Mapping Function:

Direct Mapping:

It is the simplest technique in which block j of the main memory maps onto block ‘J’ modulo 128 of the cache. Thus whenever one of the main memory blocks 0, 128, 256 is loaded in the cache, it is stored in block 0. Block 1, 129, 257 are stored in cache block 1 and so on. The contention may arise when the cache is full, when more than one

memory block is mapped onto a given cache block position. The contention is resolved by allowing the new blocks to overwrite the currently resident block. Placement of block in the cache is determined from memory address.

The memory address is divided into 3 fields, namely,

- **Low Order 4 bit field (word)** → Select one of 16 words in a block.
- **7 bit cache block field** → When new block enters cache, 7 bit determines the cache position in which this block must be stored.
- **5 bit Tag field** → The high order 5 bits of the memory address of the block is stored in 5 tag bits associated with its location in the cache.

As execution proceeds, the high order 5 bits of the address is compared with tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must be first read from the main memory and loaded into the cache. The direct mapping is shown in Figure 15.2.

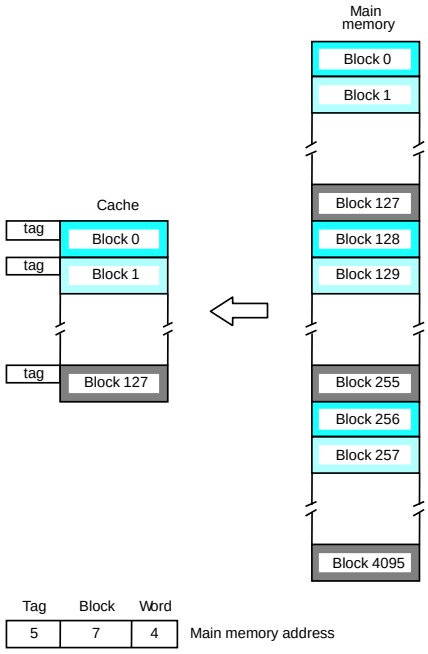


Figure 15.2: Direct Mapped Cache

Merit:

- It is easy to implement.

Demerit:

- It is not very flexible.

Associative Mapping:

Here, the main memory block can be placed into any cache block position.12 tag bits will identify a memory block when it is resolved in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called **associative mapping.** It gives complete freedom in choosing the cache location. A new block that has to be brought into the cache has to replace (eject) an existing block if the cache is full.

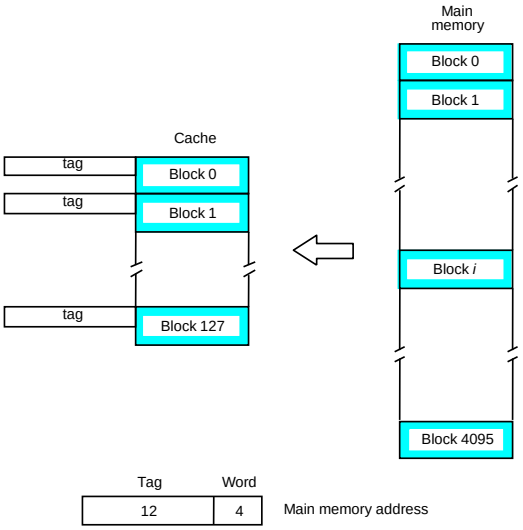


Figure 15.3: Associative Mapped Cache.

In this method, the memory has to determine whether a given block is in the cache. A search of this kind is called an associative Search. The associative-mapped cache is shown in Figure 15.3.

Merit:

- It is more flexible than direct mapping technique.

Demerit:

- Its cost is high.

Set-Associative Mapping:

It is the combination of direct and associative mapping. The blocks of the cache are grouped into sets and the mapping allows a block of the main memory to reside in any block of the specified set. In this case, the cache has two blocks per set, so the memory blocks 0, 64, 128……..4032 map into cache set to 0 and they can occupy either of the two block position within the set.

6 bit set field → Determines which set of cache contains the desired block.

6 bit tag field → the tag field of the address is compared to the tags of the two blocks of the set to clock if the desired block is present.

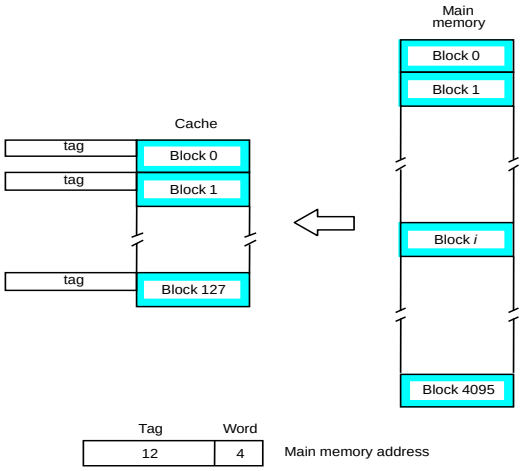


Figure 15.4: Set-Associative Mapping:

No of blocks per set	No of set field
2	6
3	5
8	4
128	no set field

The cache which contains 1 block per set is called direct Mapping. A cache that has 'k' blocks per set is called as k-way set associative cache. Each block contains a control bit called a valid bit. The Valid bit indicates that whether the block contains valid data. The dirty bit indicates that whether the block has been modified during its cache residency. The set-associative mapping is shown in Figure 15.4.

Valid bit=0→When power is initially applied to system

Valid bit =1→When the block is loaded from main memory at first time.

If the main memory block is updated by a source and if the block in the source is already exists in the cache, then the valid bit will be cleared to '0'. If Processor and DMA use the same copies of data then it is called as the Cache Coherence Problem.

Merit:

- The Contention problem of direct mapping is solved by having few choices for block placement.
- The hardware cost is decreased by reducing the size of associative search.

15.5 REPLACEMENT ALGORITHM:

In a direct-mapped cache, the position of each block is predetermined: hence, no replacement strategy exists. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. However, it is not easy to determine which blocks are about to be referenced. The property of locality of reference in programs gives a clue to a reasonable strategy. Because, programs usually stay in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LW) block, and the technique is called the LRU replacement algorithm. To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a miss occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one (It can be easily verified that the counter values of occupied blocks are always distinct). The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache. Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to

replace. Several other replacement algorithms are also used in practice. An intuitively reasonable rule would be to remove the "oldest" block from a full set when a new block must be brought in. However, because this algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove. The simplest algorithm is to randomly choose the block to be overwritten. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

Example: Let us **c**onsider 4 blocks/set, in set associative cache, where 2 bit counter can be used for each block. When a 'hit' occurs, then block counter = 0, the counter with values originally lower than the referenced one are incremented by 1 and all others remain unchanged. When a 'miss' occurs and if the set is full, the blocks with the counter value 3 is removed, the new block is put in its place and its counter is set to '0' and other block counters are incremented by 1.

Merit:

The performance of LRU algorithm is improved by randomness in deciding which block is to be overwritten.

15.6 PERFORMANCE CONSIDERATION:

Two Key factors in the commercial success are the performance and cost where the best possible performance is at low cost. A common measure of success is called the Price Performance ratio.

Performance depends on how fast the machine instructions are brought to the processor and how fast they are executed. To achieve parallelism (i.e., both the slow and fast units are accessed in the same manner) interleaving is used.

15.6.1 Interleaving:

If the main memory is structured as a collection of physically separated modules, each with its own ABR (Address buffer register) and DBR(Data buffer register), memory access operations may proceed in more than one module at the same time. Thereby the aggregate rate of transmission of words to and from the main memory system can be increased.

Two methods of address layout are indicated in Figure 15.5. In the first case, memory address generated by the processor is decoded as shown in part (a) of the figure. The high-order k bits name one of n modules and the low-order m bits name a particular word in that module. When consecutive locations are accessed, only one module is involved. At the same time, devices with DMA ability may be accessing information in other modules.

In the second case, as shown in part (b) of the figure, which is called memory interleaving. The low-order k bits of the memory address select a module, and the high-order m bits name a location within the module. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at any one time which results in both faster access to a block of data and higher average utilization of the memory system as a whole.

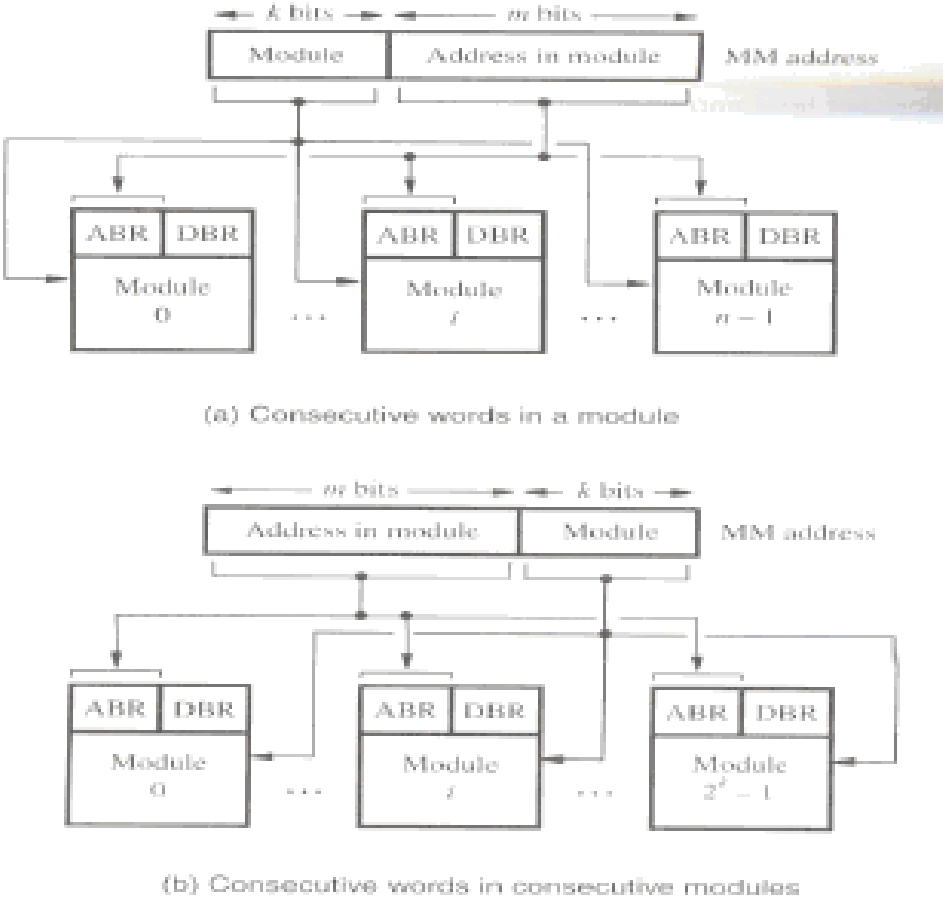


Figure 15.5: Addressing multiple-module memory system

15.6.2 HIT RATE AND MISS PENALTY

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses. Ideally, the entire memory hierarchy would appear to the CPU as a single memory unit that has the access time of a cache on the CPU chip and the size of a magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates,

over 0.9, are essential for high performance computers. Performance is adversely affected by the actions that must be taken after a miss. The extra time needed to bring the desired information into the cache is called the Miss penalty. This penalty is ultimately reflected in the time that the CPU is stalled because the required instructions or data are not available for execution. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. The miss penalty is reduced if efficient mechanisms for transferring data between the various units of the hierarchy are implemented. The previous section shows how an interleaved memory can reduce the miss penalty substantially. Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the CPU is $hC + (1 – h) M$.

15.6.3 CACHES ON PROCESSING CHIPS

When information is transferred between different chips, considerable delays are introduced in driver and receiver gates on the chips. Thus, from the speed point of view, the optimal place for a cache is on the CPU chip. Unfortunately, space on the CPU chip is needed for many other functions: this limits the size of the cache that can be accommodated. All high performance processor chips include some form of a cache. Some manufacturers have chosen to implement two separate caches, one for instructions and another for data, as in the 68040 and PowerPC 604 processors. Others have implemented a single cache for both instructions and data, as in the PowerPC 601 processor. A combined cache for instructions and data is likely to have a somewhat better hit rate, because it offers greater flexibility in mapping new information into the cache. However, if separate caches arc used, it is possible to access both caches at the same time, which leads to increased parallelism and, hence, better performance. The disadvantage of separate caches is that the increased parallelism comes at the expense of more complex circuitry. Since the Size of a cache on the CPU chip is limited by space constraints, a good strategy for designing a high performance system is to use such a cache as a primary cache. An external secondary cache, constructed with SRAM chips, is

then added to provide the desired capacity. If both primary and secondary caches are used, the primary cache should be designed to allow very fast access by the CPU, because its access time will have a large effect on the clock rate of the CPU. A cache cannot be accessed at the same speed as a register file, because the cache is much bigger and hence more complex. A practical way to speed up access in the cache is to access more than one word simultaneously and then let the CPU use them one at a time. The secondary cache can be considerably slower, but it should be much larger to ensure a high hit rate. Its speed is less critical, because it only affects the miss penalty of the primary cache. A workstation computer may include a primary cache with the capacity of tens of kilobytes and a secondary cache of several megabytes.

15.6.4 OTHER ENHANCEMENTS

In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. We discuss three of them in this section.

Write Buffer

When the write-through protocol is used, each write operation results in writing a new value into the main memory. If the CPU must wait for the memory function to be completed, as we have assumed until now, then the CPU is slowed down by all write requests. Yet the CPU typically does not immediately depend on the result of a write operation, so it is not necessary for the CPU to wait for the write request to be completed. To improve performance, a write buffer can be included for temporary storage of write requests. The CPU places each write request into this buffer and continues execution of the next instruction. The write requests stored in the write buffer are sent to the main memory whenever the memory is not responding to read requests. Note that it is important that the read requests be serviced immediately, because the CPU usually cannot proceed without the data that is to be read from the memory. Hence, these requests are given priority over write requests. The write buffer may hold a number of write requests. Thus, it is possible that a subsequent read request may refer to data that are still in the write buffer. To ensure correct operation, the addresses of data to be read

from the memory are compared with the addresses of the data in the write buffer. In case of a match, the data in the write buffer are used. This need for address comparison entails considerable cost. But the cost is justified by improved performance. A different situation occurs with the write-back protocol. In this case, the write operations are simply performed on the corresponding word in the cache. But consider what happens when a new block of data is to be brought into the cache as a result of a read miss, which replaces an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the CPU will have to wait longer for the new block to be read into the cache. It is more prudent to read the new block first. This can be arranged by providing a fast write buffer for temporary storage of the dirty block that is ejected from the cache while the new block is being read. Afterward, the contents of the buffer are written into the main memory. Thus, the write buffer also works well for the write-back protocol.

Prefetching

In the previous discussion of the cache mechanism, we assumed that new data are brought into the cache when they are first needed. A read miss occurs, and the desired data are loaded from the main memory. The CPU has to pause until the new data arrive, which is the effect of the miss penalty. To avoid stalling the CPU, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a read miss. However, the processor does not wait for the referenced data. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache by the time they are needed in the program. The hope is that prefetching will take place while the CPU is busy executing instructions that do not result in a read miss, thus allowing accesses to the main memory to be overlapped with computation in the CPU. Prefetch instructions can be inserted into a program either by the programmer or by the compiler. It is obviously preferable to have the compiler insert these instructions, which can be done with good success for many applications. Note that

software prefetching entails a certain overhead; because inclusion of prefetch instructions increases the length of programs. Moreover, some prefetches may load into the cache data that will not be used by the instruction. This can happen if the prefetched data are ejected from the cache by a read miss involving other data. However, the overall effect of software prefetching on performance is positive, and many processors (including the PowerPC) have machine instructions to support this feature. Prefetching can also be done through hardware. This involves adding circuitry that attempts to discover a pattern in memory references, and then prefetches data according to this pattern.

Lockup-Free

Cache the software prefetehing scheme just discussed does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetehing stops other accesses to the cache until the prefetch is completed. A cache of this type is said to be locked while it services a miss. We can solve this problem by modifying the basic cache structure to allow the CPU to access the cache while a miss is being serviced. In fact, it is desirable that more than one outstanding miss can be supported. A cache that can support multiple outstanding misses is called lockup-free. Since it can service only one miss at a time, it must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses. Lockup-free caches were first used in the early l980s in the Cyber series of computers manufactured by Control Data Company.

We have used software prefetching as an obvious motivation for a cache that is not locked by a read miss. A much more important reason is that, in a processor that uses a pipelined organization, which overlaps the execution of several instructions, a read miss caused by one instruction could stall the execution of other instructions. A lockup-free cache reduces the likelihood of such stalling.

Check your Progress:

1. What is a cache memory?
2. What is the importance of cache memory

3. Briefly explain the different performance considerations for caching technique.

15.7 SUMMARY

In this unit, we have discussed issues related to cache memories, their varieties, mapping functions, replacement algorithms related to caching. Also, we have discussed issues like performance and cost related to the usage and implementation of these concepts.

15.8 KEYWORDS

Cache Memory
Mapping Functions
Replacement Algorithms
Performance Considerations

15.9 ANSWER TO CHECK YOUR PROGRESS

1. 15.1
2. 15.2
3. 15.6

15.9 EXERCISES

1. What are the different mapping functions that are used in caching?
2. Discuss the set-associative-mapped method for mapping in detail.
3. Discuss the LRU replacement algorithm used in each of the mapping function in detail.
4. What are the other enhancements to improve performance.

Answer: SEE

1. 15.4
2. 15.4
3. 15.5
4. 15.6

15.10 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.
Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI. 1992

UNIT – 16: VIRTUAL MEMORIES

Structure

16.0 OBJECTIVES

After studying this module, you will be able to

- Understand the concept of virtual concept, their managements.
- Memory management by paging
- Memory management by segmentation
- Memory management requirements
- Virtual memory address translation

16.1 INTRODUCTION

Virtual memory is another important concept related to memory organization. Till now, we have assumed that the addresses generated by the processor directly specify

physical locations in the memory. This may not always be the case. For reasons that will become apparent that data may be stored in physical memory locations that have addresses different from those specified by the program. The memory control circuitry translates the address specified by the program into an address generated by the processor is referred to as a *virtual* or *logical address*. The virtual address space is mapped onto the physical memory where data are actually stored.

16.2 VIRTUAL MEMORY CONCEPT

Techniques that automatically move program and data blocks into the physical main memory when they are required for execution is called the Virtual Memory Techniques. The binary address that the processor issues either for instruction or data is called the virtual / logical address. The virtual address is translated into physical address by a combination of hardware and software components. This kind of address translation is done by MMU (Memory Management Unit). When the desired data are in the main memory, these data are fetched / accessed immediately. If the data are not in the main memory, The MMU causes the operating system to bring the data into memory from the disk. Transfer of data between disk and main memory is performed using DMA scheme.

Figure 16.1: Virtual Memory Organization

16.3 MEMORY MANAGEMNT BY PAGING:

In Paged memory management, each job's address space is divided into equal size pieces called pages, and likewise physical memory is divided into equal sized pieces of the same size as a page called blocks. Then by providing a suitable hardware mapping facility, any page can be mapped onto any block. The pages remain logically contiguous, but the corresponding blocks are not contiguous.

For the hardware to perform the mapping from address space to physical memory there must be a separate register for each page, these registers are called as page maps or Page Map Tables (PMTs).

Page Table:

It contains the information about the main memory address where the page is stored and the current status of the page.

Page Frame:

An area in the main memory that holds one page is called the page frame.

Page Table Base Register:

It contains the starting address of the page table.

Virtual Page Number + Page Table Base register → Gives the address of the corresponding entry in the page table i.e., it gives the starting address of the page if that page currently resides in memory.

16.4 MEMORY MANAGEMENT BY SEGMENTATION:

Memory segmentation is the division of computer's primary memory into **segments** or **sections**. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset within that segment. Segments or sections are also used in object files of compiled programs when they are linked together into a program image and when the image is loaded into memory

16.5 VIRTUAL MEMORY ADDRESS TRANSLATION

In address translation, all programs and data are composed of fixed length units called Pages. The Page consists of a block of words that occupy contiguous locations in the main memory. The pages are commonly range from 2K to 16K bytes in length.

The cache bridge speeds up the gap between main memory and secondary storage and it is implemented in software techniques. Each virtual address generated by the processor contains virtual page number (Low order bit) and offset (High order bit)

Virtual Page number + Offset → this specifies the location of a particular byte (or word) within a page.

Page Table:

It contains the information about the main memory address where the page is stored and the current status of the page.

Page Frame:

An area in the main memory that holds one page is called the page frame.

Page Table Base Register:

It contains the starting address of the page table.

Virtual Page Number + Page Table Base register → Gives the address of the corresponding entry in the page table i.e., it gives the starting address of the page if that page currently resides in memory.

Control Bits in Page Table:

The Control bits specify the status of the page while it is in main memory.

Function:

The control bit indicates the validity of the page, i.e. it checks whether the page is actually loaded in the main memory. It also indicates that whether the page has been modified during its residency in the memory; this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page.

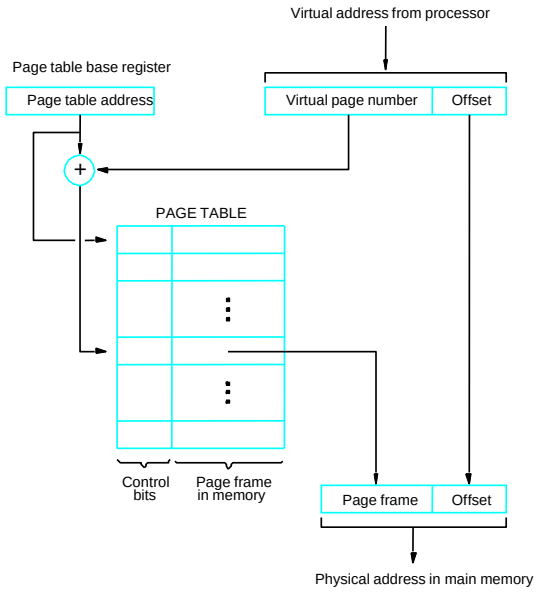


Figure 16.2: Virtual Memory Address Translation

The Page table information is used by MMU for every Read and Write access. The Page table is placed in the main memory but a copy of the small portion of the page table is located within MMU. This small portion or small cache is called Translation Look Aside Buffer (TLB). This portion consists of the page table entries that corresponds to the most recently accessed pages and also contains the virtual address of the entry.

When the operating system changes the contents of page table, the control bit in TLB will invalidate the corresponding entry in the TLB. Given a virtual address, the MMU looks in TLB for the referenced page. If the page table entry for this page is found in TLB, the physical address is obtained immediately. If there is a miss in TLB, then the required entry is obtained from the page table in the main memory and TLB is updated. When a program generates an access request to a page that is not in the main memory, then Page Fault will occur. The whole page must be brought from disk into memory before an access can proceed. When it detects a page fault, the MMU asks the operating system to generate an interrupt.

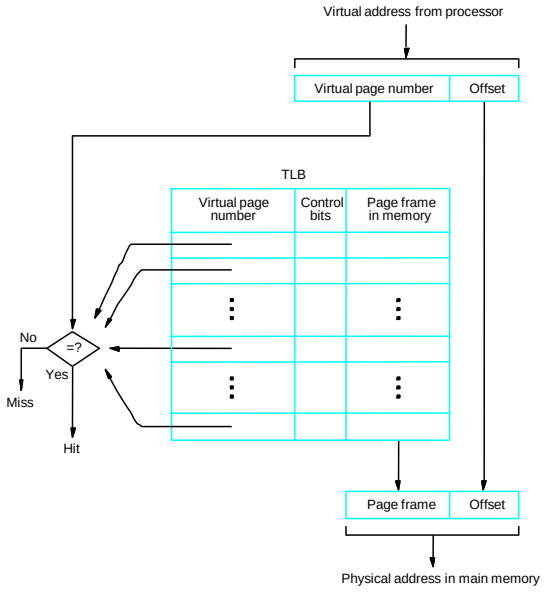


Figure 16.3: Use of Associative Mapped TLB

The operating System suspends the execution of the task that caused the page fault, and then begins execution of another task whose pages are in main memory because the long delay occurs while page transfer takes place. When the task resumes, either the interrupted instruction must continue from the point of interruption or the instruction must be restarted. If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. In that case, it uses LRU algorithm which removes the least referenced page.

A modified page has to be written back to the disk before it is removed from the main memory. In that case, write – through protocol is used.

MEMORY MANAGEMENT REQUIREMENTS:

Memory management routines are part of the Operating system. Assembling the OS routine into virtual address space is called '**System Space'**. The virtual space in which the user application program resides is called the **'User Space'**. Each user space has a separate page table.

The MMU uses the page table to determine the address of the table to be used in the translation process. Hence by changing the contents of this register, the OS can switch from one space to another. The process has two stages. They are User State and Supervisor state.

User State:

In this state, the processor executes the user program.

Supervisor State:

When the processor executes the operating system routines, the processor will be in supervisor state.

Privileged Instruction:

In user state, some machine instructions cannot be executed. Hence a user program is prevented from accessing the page table of other user spaces or system spaces. The control bits in each entry can be set to control the access privileges granted to each program. One program may be allowed to read/write a given page, while the other programs may be given only read access.

16.6 SUMMARY

In this unit, we have discussed a concept which is so very evident and useful called the virtual memory concept. We explained very popular Memory Management Techniques, namely, Paging and Segmentation. We also discussed about Virtual Memory Address Translation.

Check your progress:

1. What do you mean by virtual memory concept?
2. What are Virtual Memory Techniques?

3. What is page table?

16.7 KEYWORDS

Virtual Memory,

16.8 ANSWERS TO CHECK YOUR PROGRESS

1. 16.1
2. 16.2
3. 16.3

16.8 UNIT-END EXERCISES AND ANSWERS

1. What is a virtual memory?
2. How is address translation done? Explain in detail.
3. Discuss the various memory management requirements in virtual memory concept.

Answers: SEE

1. 16.2
2. 16.5
3. 16.5

16.9 SUGGESTED READINGS

Text Book:

Computer Organization – Carl Hamacher, ZvonkoVranesic, SafwatZaky, MGH publications, Fifth Edition, 2002.

Reference Books:

Digital logic and computer design: Morris Mano, PHI, 23rd Reprint, October 2000.

Ronald J Toci, Digital Systems – Principles and Applications, 5th edition, PHI. 1992