DIGITAL NOTES

ON

COMPUTER ORGANIZATION AND ARCHITECTURE

B.TECH II YEAR - I SEM (2022-2023)

PREPARED BY:

MADDELA SAI KIRAN

ASSISTANT PROFESSOR

Dept. of CSE-AIML

UNIT – I

Unit-1

Functional blocks of a computer: CPU, memory, input-output subsystems, control unit. Instruction set architecture of a CPU – registers, instruction execution cycle, RTL interpretation of instructions, addressing modes, instruction set. Case study – instruction sets of some common CPUs

Functional Units

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1.

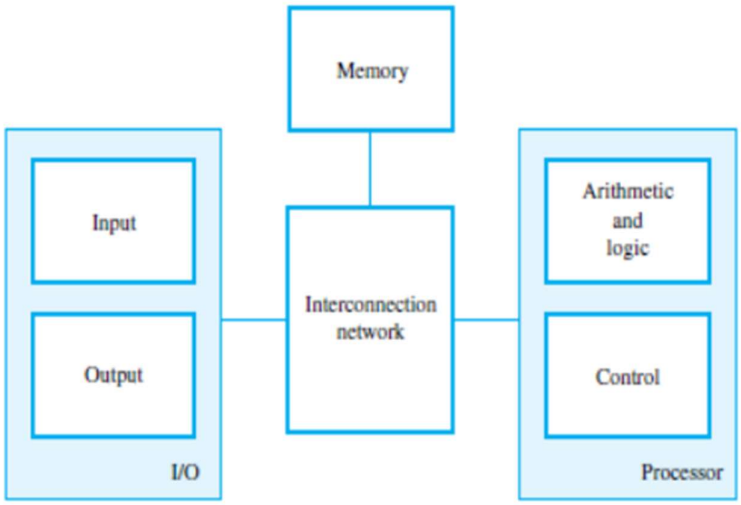


Figure 1.1 Basic functional units of a computer.

The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer‟s memory, either for later use or to be processed immediately by the arithmetic and logic unit.

The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit.

All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.

The arithmetic and logic circuits, in conjunction with the main control circuits, is the processor. Input and output equipment is often collectively referred to as the input-output (I/O) unit.

A program is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations.

The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it.

Data are numbers and characters that are used as operands by the instructions. Data are also stored in the memory. The instructions and data handled by a computer must be encoded in a suitable format. Each instruction, number, or character is encoded as a string of binary digits called bits, each having one of two possible values, 0 or 1, represented by the two stable states.

Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input. Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed.

The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called words.

The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits. To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations.

Instructions and data can be written into or read from the memory under the control of the processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM).

The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The

cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache.

When the execution of an instruction requires data, located in the main memory, the data are fetched and copies are also placed in the cache. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use.

Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondarystorage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.

Access times for secondary storage are longer than for primary memory. The devices available are including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit

Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.

When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

Output unit function is to send processed results to the outside world. A familiar example of such a device is a printer. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute.

However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor. Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases.

Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred.

Control circuits are responsible for generating the timing signals that govern the transfers. They determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals.

A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

• The computer accepts information in the form of programs and data through an input unit and stores it in the memory.

• Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.

• Processed information leaves the computer through an output unit.

• All activities in the computer are directed by the control unit.

REGISTERS

A register is a group of flip flops with each flip-flop capable of storing one bit of information.

An n-bit register has a group of n-flip flops and is capable of storing any binary information of n bits.

Register Symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

The computer registers are designated by capital letters to denote the function of the register.

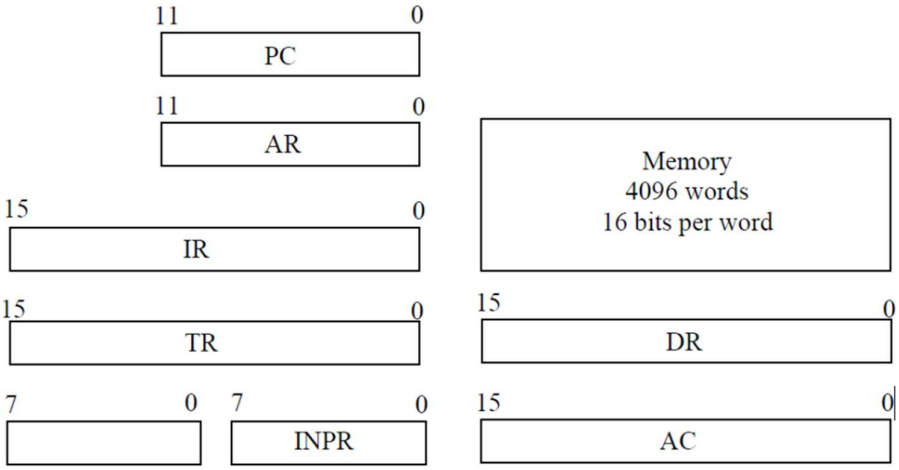


Figure 1-3 Basic computer registers and memory.

Computer Instruction

• The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as

they appear in memory words or in a control register.

• The bits of the instruction are divided into groups called fields.

• The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.

2. An address field that designates a memory address or a processor register.

3. A mode field that specifies the way the operand or the effective address is determined.

Mode field	Opcode field	Address field

Computer Instruction

• The basic computer has three instruction code formats, each format has 16 bits.

• The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.

• 1. Memory reference instruction

• 2. Register-reference instruction

• 3. Input-Output instruction

• A memory reference:(opcode=000 through 110, I=0 or 1) instruction uses 12 bits to specify an address and

one bit to specify the addressing mode and 3 bits specify an opcode

• I is equal to 0 for direct address and to 1 for indirect address.

• The register-reference: (opcode=111, I=0) instructions are recognized by the operation code 111 with a 0 in

the leftmost bit (bit 15) of the instruction.

• A register-reference instruction specifies an operation on or a test of the AC register. An operand from

memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.

• Input-Output instruction: (opcode=111, I=1)

mode field opcode field Address field

INSTRUCTION CYCLE

• A program residing in the memory unit of the computer consists of a sequence of instructions.

• The program is executed in the computer by going through a cycle for each instruction. Each

instruction cycle in turn is subdivided into a sequence of sub cycles or phases.

• In the basic computer each instruction cycle consists or the following phases:

• 1. Fetch an instruction from memory.

• 2. Decode the instruction

• 3. Read the effective address from memory if the instruction has an indirect address.

• 4. Execute the instruction.

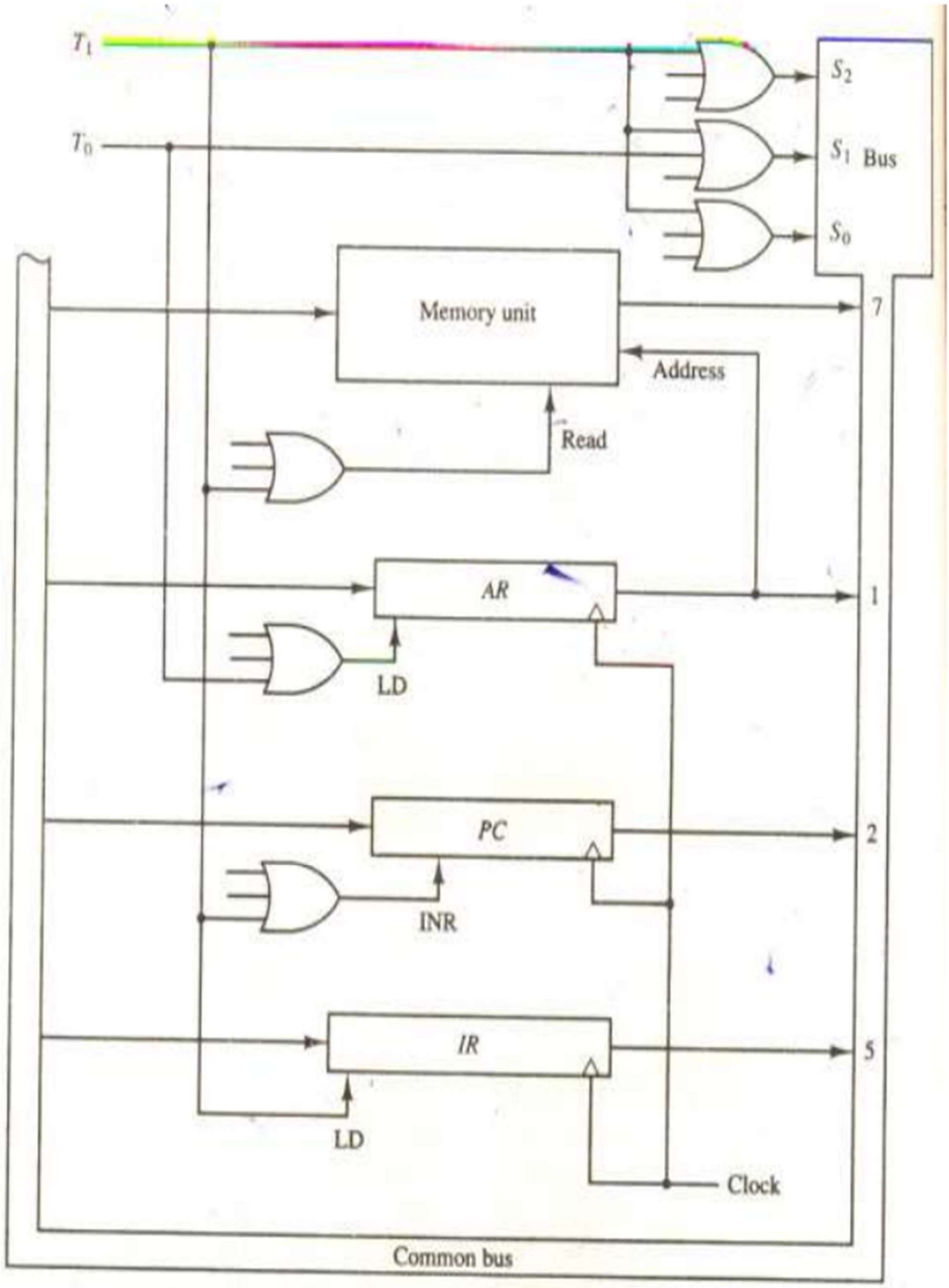
FETCH AND DECODE

• T0: AR ← PC

• T1: IR ← M[AR], PC ← PC + 1

• T2: D0,…, D1 ← Decode IR (12-14), AR ← IR(0 −11), 1 ← IR (15)

INSTRUCTION CYCLE DIAGRAM



INSRTUCTION EXECUTION STEPS:

• The Above Figure shows how the first two register transfer statements are implemented in the bus system.

• To provide the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection.:

• 1. Place the content of PC onto the bus by making the bus selection inputs S2 S1 S0 equal to 010.

• 2. Transfer the content of the bus to AR y enabling the LD input of AR.

• The next clock transition initiates the transfer form PC to AR since T0 = 1.

• In order to implement the second statement

• T1: IR ← M[AR], PC ← PC + 1

• It is necessary to use timing signal T1 to provide the following connections in the bus system.

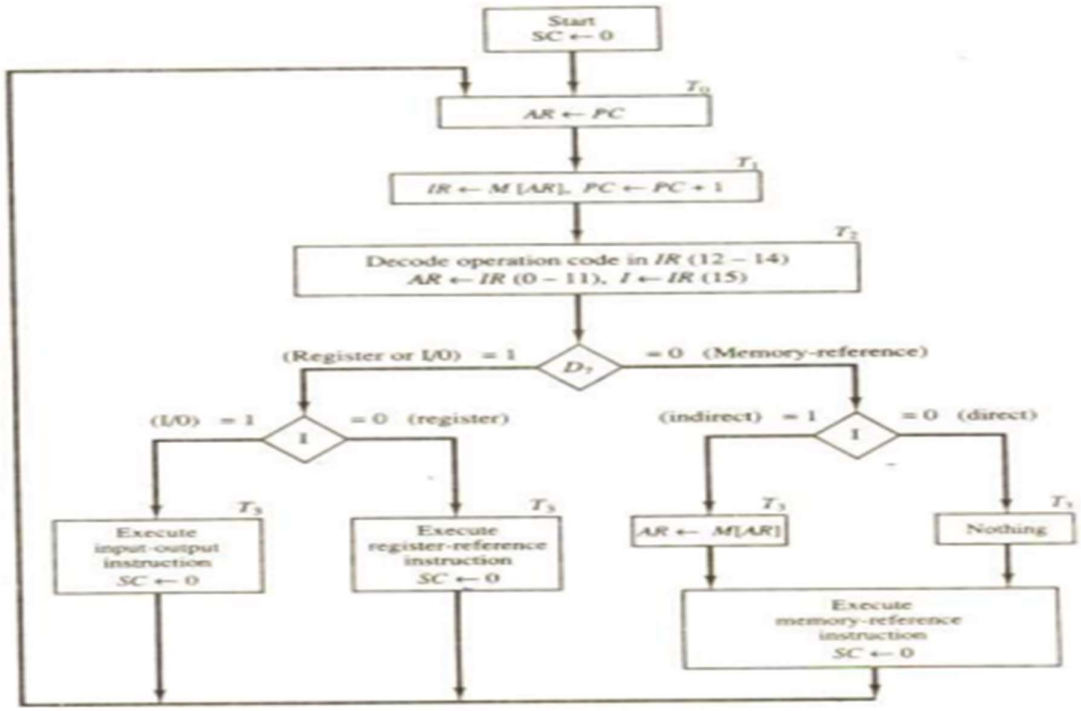
• 1. Enable the read input of memory.

• 2. Place the content of memory onto the bus by making S2 S1S0 = 111.

• 3. Transfer the content of the bus to IR by enabling the LD input of IR.

• 4. Increment PC by enabling the INR input of PC.

FLOW CHART OF INSTRUCTION CYCLE



ADDRESSING MODES

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words.

The way the operands are chosen during program execution in dependent on the addressing mode of the instruction.

The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to Memory, counters for loop control, indexing of data, and program relocation

2. To reduce the number of bits in the addressing field of the instruction.

3.The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

Types of addressing modes

1. Implied Addressing Mode.

2. Immediate Addressing Mode

3. Register Addressing Mode

4. Register Indirect Addressing Mode

5. Auto-increment or Auto-decrement Addressing Mode

6. Direct Addressing Mode

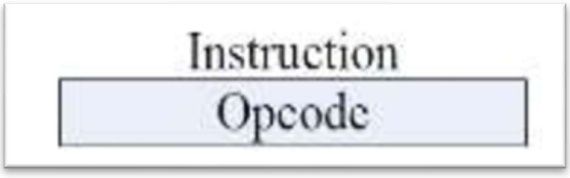
7. Indirect Addressing Mode

8. Relative Addressing Mode

9. Index Addressing Mode

10. Base Register Addressing Mode

Implied Addressing Mode:



• In this mode the operands are specified implicitly in the definition of the instruction.

• For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.

• Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

• Advantage: no memory reference.

• Disadvantage: limited operand.

Immediate Addressing Mode:

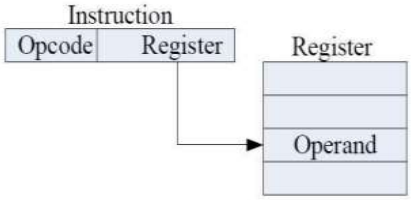
• In this mode the operand is specified in the instruction itself.

• In other words, an immediate-mode instruction has an operand field rather than an address field.

• For example MVI B, 50H

Register Addressing Mode:

• In this mode, the operands are in registers that reside within the CPU. The particular register is selected from the register field in the instruction. A k-bit field can specify any one of 2k registers



• Effective Address (EA) = R

• Advantage: no memory reference.

• Disadvantage: limited address space

Auto-increment or Auto-decrement Addressing Mode:

• This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

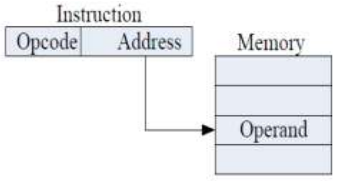
• When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

• This can be achieved by using the increment or decrement instruction.

Direct Address Mode:

• In this mode the effective address is equal to the address part of the instruction.

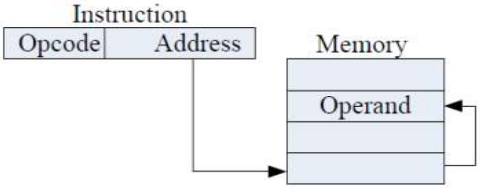
• The operand resides in memory and its address is given directly by the adder field of the instruction.



Indirect Address Mode:

• In this mode the address field of the instruction gives the address where the effective address is stored in memory.

• Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.



Effective address=address part of the instruction + content of CPU register

Relative Addressing Mode:

• In this mode the content of the program counter (PC) is added to the address part of the instruction in order to obtain the effective address.

• The address part of the instruction is usually a signed number (either a +ve or a –ve number).

• When the number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

Effective address= address part of the instruction + content of the PC.

Indexed Addressing Mode:

• In this mode the content of an index register (XR) is added to the address part of the instruction to obtain the effective address.

• The index register is a special CPU register that contains an index value.

• Note: If an index-type instruction does not include an address field in its format, the instruction is automatically converted to the register indirect mode of operation.

Effective Address (EA) = Content of indexed register(XR) + Address part of the instruction.

Base Register Addressing Mode :

• In this mode the content of a base register (BR) is added to the address part of the instruction to obtain the effective address.

• This is similar to the indexed addressing mode except that the register is now called a base register instead of the index register.

Effective Address (EA) = Content of indexed register(BR) + Address part of the instruction.

Numerical Example

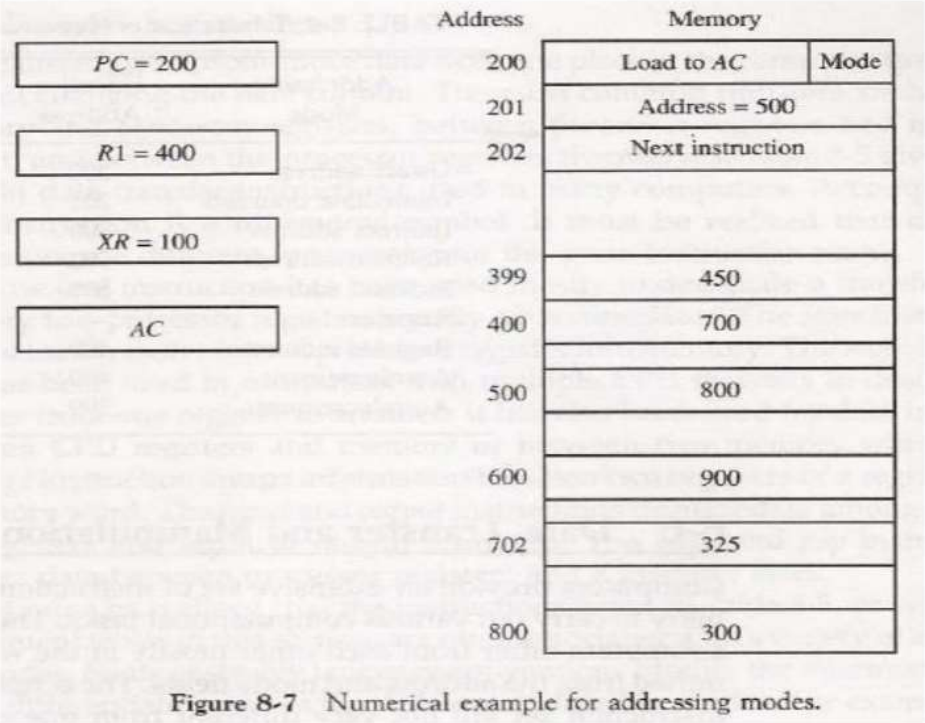


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Register Transfer Language And Micro Operations:

Register Transfer language:

 Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic.

 The modules are interconnected with common data and control paths to form a digital computer system.

 The operations executed on data stored in registers are called microoperations.

 A microoperation is an elementary operation performed on the information stored in one or more registers

  Examples are shift, count, clear, and load

  Some of the digital components from before are registers that implement microoperations.

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

  The set of registers it contains and their functions

  The sequence of microoperations performed on the binary information stored

 The control that initiates the sequence of microoperations

Use symbols, rather than words, to specify the sequence of microoperations. The symbolic notation used is called a register transfer language.

A programming language is a procedure for writing symbols to specify a given computational process Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations.

Register Transfer

Designate computer registers by capital letters to denote its function.

The register that holds an address for the memory unit is called MAR.

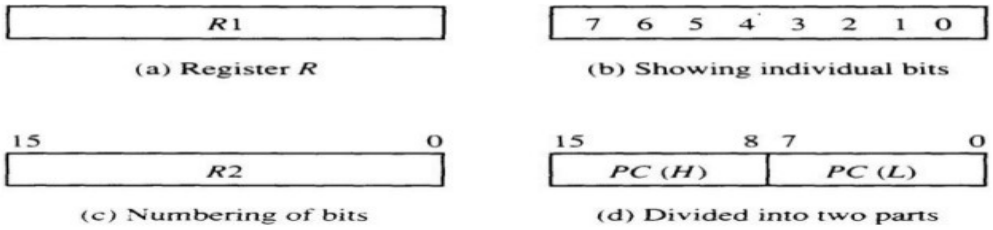
The program counter register is called PC.

IR is the instruction register and R1 is a processor register.

The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1.

Refer to Figure 4.1 for the different representations of a register

Figure 4-1 Block diagram of register.



 Designate information transfer from one register to another by R2 R1

 This statement implies that the hardware is available

 The outputs of the source must have a path to the inputs of the destination

 The destination register has a parallel load capability

 If the transfer is to occur only under a predetermined control condition, designate it by

If (P = 1) then (R2 R1) or, P: R2 R1,where P is a control function that can be either 0 or 1

 Every statement written in register transfer notation implies the presence of the required hardware construction.

Figure 4-2 Transfer from R1 to R2 when P = 1.

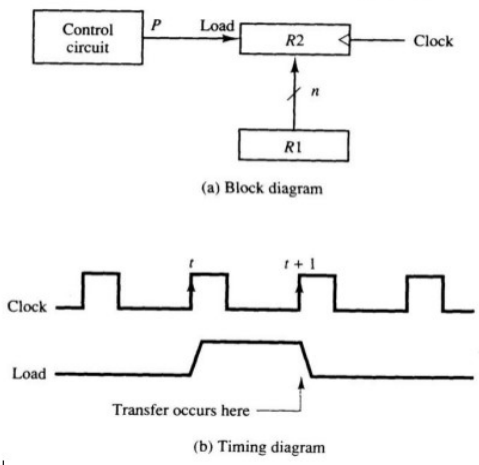


TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0-7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

INSTRUCTION FORMATS

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities. They list all hardware-implemented instructions, specify their binary code format, and provide a precise definition of each instruction.

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1) An operation code field that specifies the operation to be performed.

2) An address field that designates a memory address or a processor registers.

3) A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. The various addressing modes that have been formulated for digital computers are presented in Sec. 5.5. In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields is an instruction.

Operations specified by computer instructions are executed on some data stored in memory or processor registers, Operands residing in processor registers are specified with a register

address. A register address is a binary number of k bits that defines one of 2k registers in the CPU.

Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

Computers may have instructions of several different lengths containing varying number of addresses. The number

of address fields in the instruction format of a computer depends on the internal organization of its registers. Most

computers fall into one of three types of CPU organizations:

1 Single accumulator organization.

2 General register organization.

3 Stack organization.

An example of an accumulator-type organization is the basic computer presented in Chap. 5. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD.

Where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M[X].

AC is the accumulator register and M[X] symbolizes the memory word located at address X.

An example of a general register type of organization was presented in Fig. 7.1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in anassembly language as ADD R1, R2, R3

To denote the operation R1 ← R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction ADD R1, R2 Would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction.

Thus the instruction MOV R1, R2 Denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer).

Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by ADD R1, X Would specify the operation R1 ← R + M [X].

It has two address fields, one for register R1 and the other for the memory address X.

The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field.

Thus the instruction PUSH X Will push the word at address X to the top of the stack. The stack pointer is updated automatically.

Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack.

The instruction ADD In a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.

There is no need to specify operands with an address field since all operands are implied to be in the stack.

Most computers fall into one of the three types of organizations that have just been described. Some computers combine features from more than one organization structure. For example, the Intel 808- microprocessor has seven CPU registers, one of which is an accumulator register. As a consequence; the processor has some of the characteristics of a general register type and some of the characteristics of a accumulator type.

All arithmetic and logic instruction, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields.

Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack-organized CPU.

To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic

statement $X = (A + B) * (C + D)$.

Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations;

MOV for the transfer-type operation; and **LOAD** and **STORE** for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

THREE-ADDRESS INSTRUCTIONS

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates $X = (A + B) * (C + D)$ is shown below, together with comments that explain the register transfer operation of each instruction.

ADD R1, A, B R1 ← M [A] + M [B]

ADD R2, C, D R2 ← M [C] + M [D]

MUL X, R1, R2 M [X] ← R1 * R2

It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.

The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170.

The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

TWO-ADDRESS INSTRUCTIONS

Two address instructions are the most common in commercial computers. Here again each address field can specify either

a processor register or a memory word. The program to evaluate X = (A + B) ∗ (C + D) is as follows:

MOV R1, A R1 ← M [A]

ADD R1, B R1 ← R1 + M [B]

MOV R2, C R2 ← M [C]

ADD R2, D R2 ← R2 + M [D]

MUL R1, R2 R1 ← R1∗R2

MOV X, R1 M [X] ← R1

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

ONE-ADDRESS INSTRUCTIONS

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second and assume that the AC contains the result of tall operations. The program to evaluate X = (A + B) ∗ (C + D) is

LOAD A AC ← M [A]

ADD B AC ← A [C] + M [B]

STORE T M [T] ← AC

LOAD C AC ← M [C]

ADD D AC ← AC + M [D]

MUL T AC ← AC ∗ M [T]

STORE X M [X] ← AC

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

ZERO-ADDRESS INSTRUCTIONS

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.

The following program shows how X = (A + B) ∗ (C + D) will be written for a stack organized computer. (TOS stands for top of stack)

PUSH A TOS ← A

PUSH B TOS ← B

ADD TOS ← (A + B)

PUSH C TOS ← C

PUSH D TOS ← D

ADD TOS ← (C + D)

MUL TOS ← (C + D) ∗ (A + B)

POP X M [X] ← TOS

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

UNIT – II

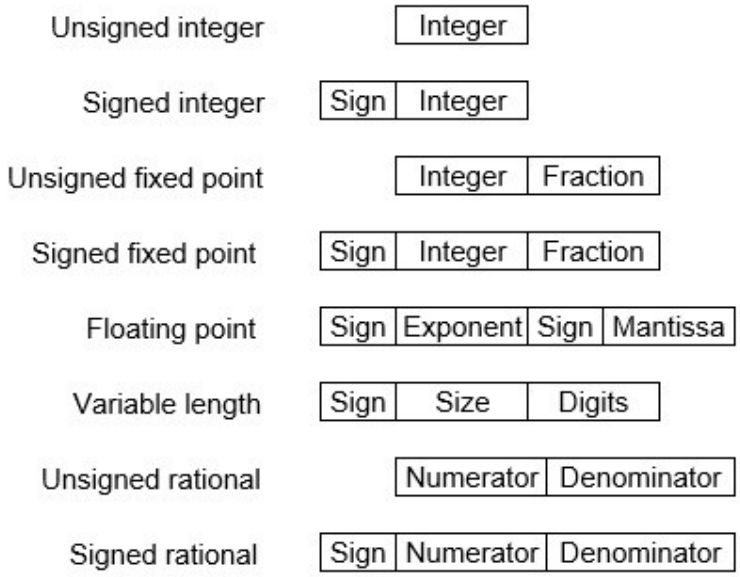
Fixed Point and Floating Point Number Representations:

Digital Computers use Binary number system to represent all types of information inside the computers. Alphanumeric characters are represented using binary bits (i.e., 0 and 1). Digital representations are easier to design, storage is easy, accuracy and precision are greater.

There are various types of number representation techniques for digital number representation, for example: Binary number system, octal number system, decimal number system, and hexadecimal number system etc. But Binary number system is most relevant and popular for representing numbers in digital computer system.

Storing Real Number:

These are structures as following below −



There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing. These are (i) Fixed Point Notation and (ii) Floating Point Notation. In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

Fixed-Point Representation −

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

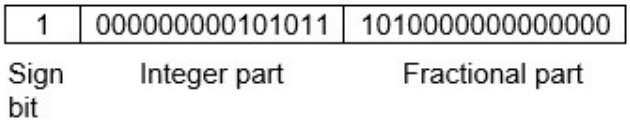
We can represent these numbers using:

- Signed representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 1's complement representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 2's complementation representation: range from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

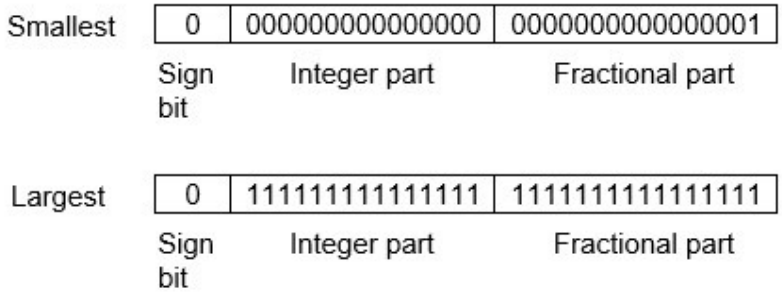
Example −Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.



These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx 0.000015$ approximate and the largest positive number is $(2^{15}-1)+(1-2^{-16})=2^{15}(1-2^{-16}) =32768$, and gap between these numbers is 2^{-16}.

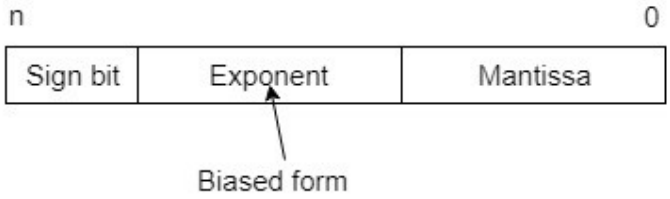
We can move the radix point either left or right with the help of only integer field is 1.

Floating-Point Representation −

This representation does not reserve a specific number of bits for the integer part or the fractional part. Instead it reserves a certain number of bits for the number (called the mantissa or significand) and a certain number of bits to say where within that number the decimal place sits (called the exponent).

The floating number representation of a number has two part: the first part represents a signed fixed point number called mantissa. The second part of designates the position of the decimal (or binary) point and is called the exponent. The fixed point mantissa may be fraction or an integer. Floating -point is always interpreted to represent a number in the following form: Mxr^e.

Only the mantissa m and the exponent e are physically represented in the register (including their sign). A floating-point binary number is represented in a similar manner except that is uses base 2 for the exponent. A floating-point number is said to be normalized if the most significant digit of the mantissa is 1.



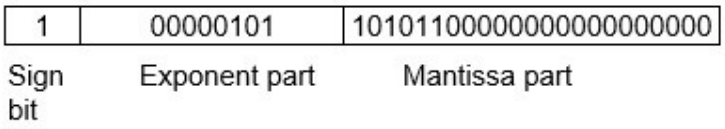
So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number.

Note that signed integers and exponent are represented by either sign representation, or one's complement representation, or two's complement representation.

The floating point representation is more flexible. Any non-zero number can be represented in the normalized form of $\pm(1.b_1b_2b_3 ...)_2x2^n$ This is normalized form of a number x.

Example −Suppose number is using 32-bit format: the 1 bit sign bit, 8 bits for signed exponent, and 23 bits for the fractional part. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a *"hidden bit*".

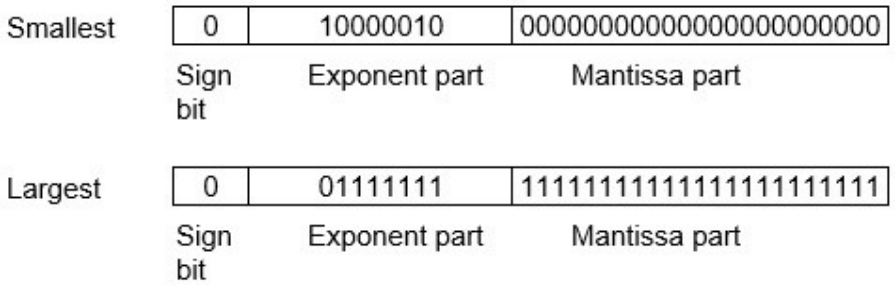
Then −53.5 is normalized as $-53.5=(-110101.1)_2=(-1.101011)x2^5$, which is represented as following below,



Where 00000101 is the 8-bit binary value of exponent value +5.

Note that 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.

The smallest normalized positive number that fits into 32 bits is $(1.00000000000000000000000)_2 x 2^{-126} = 2^{-126} \approx 1.18 x 10^{-38}$, and largest normalized positive number that fits into 32 bits is $(1.11111111111111111111111)_2 x 2^{127} = (2^{24}-1) x 2^{104} \approx 3.40 x 10^{38}$. These numbers are represented as following below,



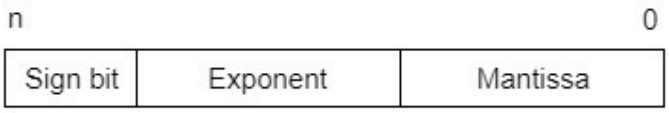
The precision of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

The gap between 1 and the next normalized floating-point number is known as machine epsilon. the gap is $(1+2^{-23})-1=2^{-23}$ for above example, but this is same as the smallest positive floating-point number because of non-uniform spacing unlike in the fixed-point scenario.

Note that non-terminating binary numbers can be represented in floating point representation, e.g., $1/3 = (0.010101\ ...)_2$ cannot be a floating-point number as its binary representation is non-terminating.

IEEE Floating point Number Representation −

IEEE (Institute of Electrical and Electronics Engineers) has standardized Floating-Point Representation as following diagram.



So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number. The sign bit is 0 for positive number and 1 for negative number. Exponents are represented by or two's complement representation.

According to IEEE 754 standard, the floating-point number is represented in following ways:

- Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa
- Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
- Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
- Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

BINARY ADDER

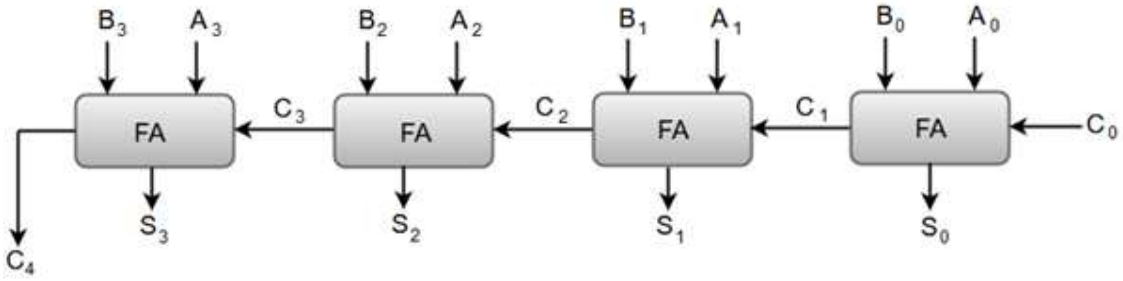
The Add micro-operation requires registers that can hold the data and the digital components that can perform the arithmetic addition.

A Binary Adder is a digital circuit that performs the arithmetic sum of two binary numbers provided with any length.

A Binary Adder is constructed using full-adder circuits connected in series, with the output carry from one full-adder connected to the input carry of the next full-adder.

The following block diagram shows the interconnections of four full-adder circuits to provide a 4-bit binary adder.

4 bit binary adder:



- The augend bits (A) and the addend bits (B) are designated by subscript numbers from right to left, with subscript '0' denoting the low-order bit.
- The carry inputs starts from C0 to C3 connected in a chain through the full-adders. C4 is the resultant output carry generated by the last full-adder circuit.
- The output carry from each full-adder is connected to the input carry of the next-high-order full-adder.
- The sum outputs (S0 to S3) generates the required arithmetic sum of augend and addend bits.
- The ***n*** data bits for the **A** and **B** inputs come from different source registers. For instance, data bits for **A** input comes from source register R1 and data bits for **B** input comes from source register R2.
- The arithmetic sum of the data inputs of A and B can be transferred to a third register or to one of the source registers (R1 or R2).

Carry Look Ahead Adder:

A carry look-ahead adder (CLA) is an electronic adder used for binary addition. Due to the quick additions performed, it is also known as a fast adder. The CLA logic uses the concepts of generating and propagating carries. We can say that the CLA adder is the successor of the Ripple Carry Adder.

Working of a Carry Look Ahead Adder:

Let us consider a full adder. We have the inputs signals A, B, and Cin. If we consider the addition of these three variables in every possible case, we get a truth table like the one below.

Truth Table of Full Adder

Truth Table of Full Adder

On analyzing the truth table, we see that the Carry is 1 when

Either the value of A or B is one, as well as Cin, is 1, or

Both A and B have the value 1.

Let us now consider two new variables, Carry Generate (Gi) and Carry Propagate (Pi).

For case 1, we see that an output carry is propagated, when we give an input carry. We will refer to this with Pi. So, the mathematical expression of Pi can we represented as :

Pi = Ai ⊕ Bi

While considering case 2, we see that an output carry is generated when both inputs, A and B, are high, regardless of the value of the input carry. We will refer to this output carry as Gi. Thus, we can mathematically express Gi as :

Gi = Ai . Bi

Originally, for a full adder we have the following equations:

Sum = A⊕B⊕Ci

Carry = Ci(A+B) + AB

Thus, we can rewrite the equations of the full adder in terms of Carry Propagate (Pi) and Carry Generate (Gi) as :

Sum = Pi ⊕ Ci

Carry = Gi + Pi . Ci

The equations of Sum and Carry can be represented by a logic circuit given below.

Circuit using logic gates

Logic Circuit

4-bit Carry Look Ahead Adder

Consider the 4-bit Carry Look Ahead Adder system shown below.

4-bit Carry Look Ahead Adder

4-bit Carry Look Ahead Adder

We can calculate the output carry C1, C2, C3, and C4 using the above derived equations as:

C1 = (Cin . P0) + G0

C2 = (C1 . P1) + G1 = (((Cin . P0) + G0) . P1) + G1

= (Cin . P0 . P1) + (G0 . P1) + G1

C3 = (C2 . P2) + G2 = (((C1 . P1) + G1) . P2) + G2

= G2 + (P2 . G1) + (P2 . P1 . G0) + (P2 . P1 . P0 . Cin)

C4 = (C3 . P3) + G3

= (Cin . P0 . P1 . P2 . P3) + (P3 . P2 . P1 . G0) + (P3 . P2 . G1) + (G2 . P3) + G3

Circuit Diagram

The circuit for the above equations can be constructed as shown below.

Circuit Diagram of 4-bit CLA Adder

Circuit Diagram of 4-bit Carry-Lookahead Adder

Circuit Diagram of the entire 4-bit CLA Adder

Circuit Diagram of the entire 4-bit CLA Adder

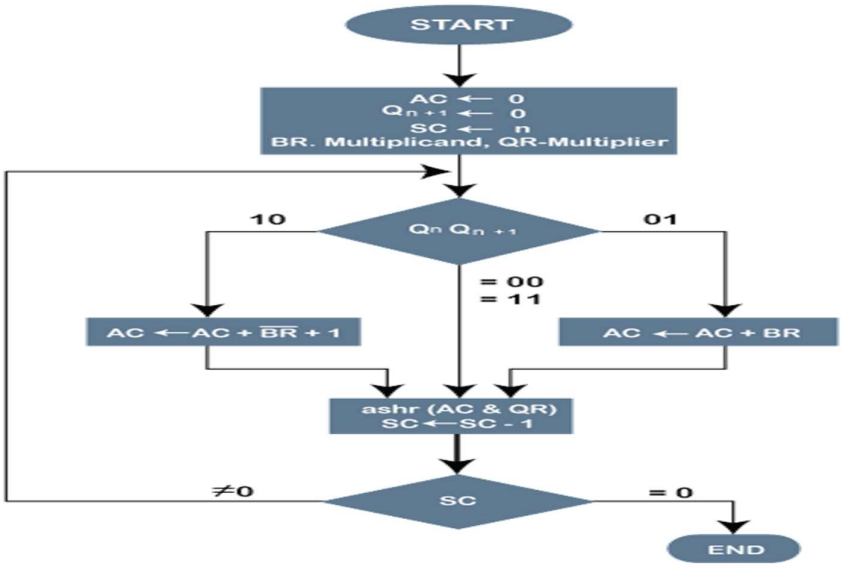
We can see that there is no dependency on any intermediate Carry values in any of the equations. On solving the equations, we see that only the input Carry Cin is required to calculate all the Sum and Output Carry values. This resolves the issue faced by the Ripple Carry Adder's dependency on the intermediate carry values.

Thus, the entire operation works faster for higher-order bits, when compared to the Ripple Carry Adder. This is the reason why the CLA Adder is also called as a Fast Adder.

Booth's Multiplication Algorithm

The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2^k to weight 2^m that can be considered as $\mathbf{2^{k+1} - 2^m}$.

Following is the pictorial representation of the Booth's Algorithm:



In the above flowchart, initially, **AC** and $\mathbf{Q_{n+1}}$ bits are set to 0, and the **SC** is a sequence counter that represents the total bits set **n,** which is equal to the number of bits in the multiplier. There are **BR** that represent the **multiplicand bits,** and QR represents the **multiplier bits**. After that, we encountered two bits of the multiplier as Q_n and Q_{n+1}, where Qn represents the last bit of QR, and Q_{n+1} represents the incremented bit of Qn by 1. Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ashr). If the

two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (**ashr**), including $\mathbf{Q_{n+1}}$. The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged. And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).

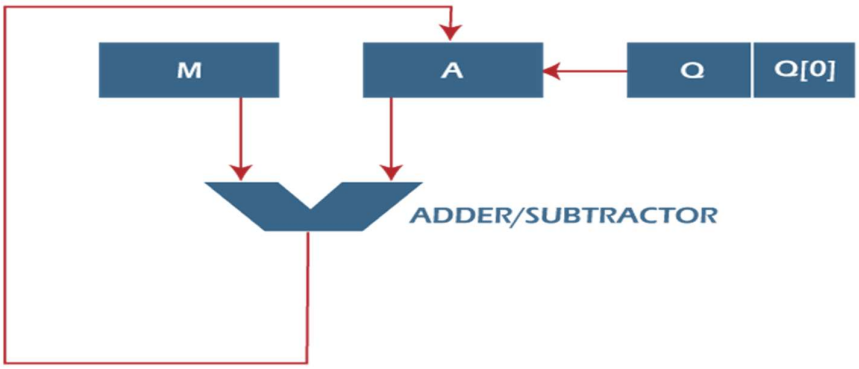
Working on the Booth Algorithm

1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and Q_{n+1} registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A Qn represents the last bit of the Q, and the Q_{n+1} shows the incremented bit of Qn by 1.
5. On each cycle of the booth algorithm, Q_n and Q_{n+1} bits will be checked on the following parameters as follows:
    i. When two bits Q_n and Q_{n+1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of Qn and Q_{n+1} is incremented by 1 bit.
    ii. If the bits of Q_n and Q_{n+1} is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
    iii. If the bits of Q_n and Q_{n+1} is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
6. The operation continuously works till we reached n - 1 bit in the booth algorithm.
7. Results of the Multiplication binary bits will be stored in the AC and QR registers.

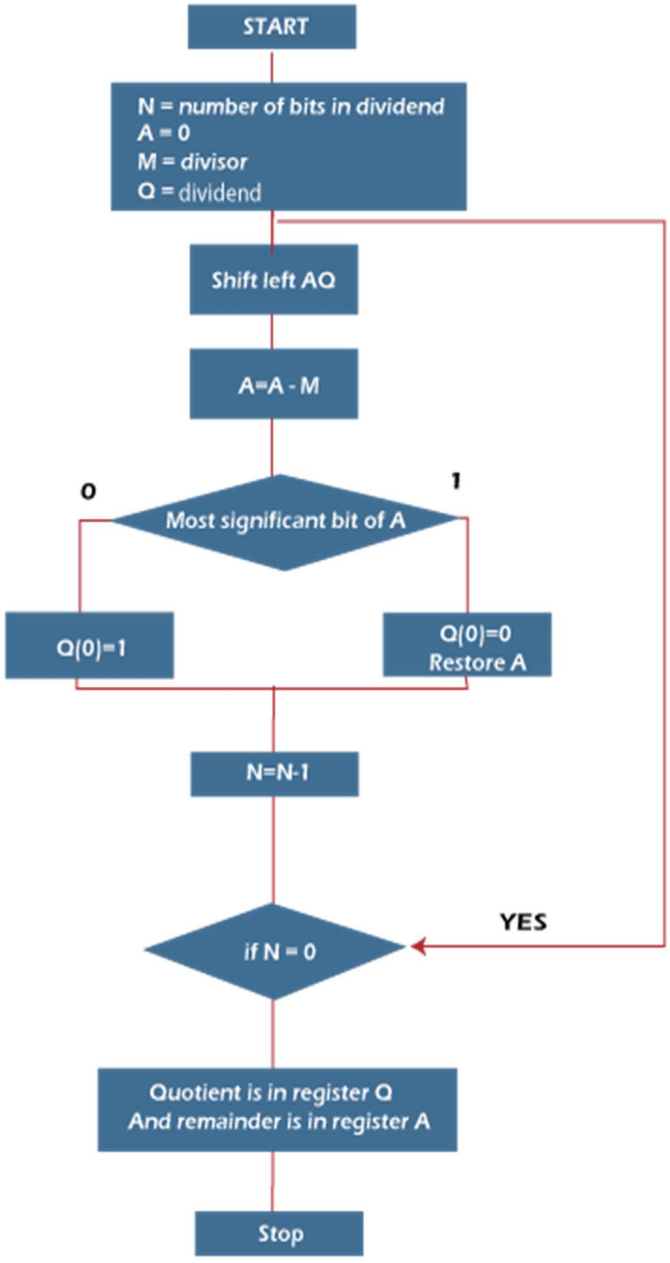
Restoring Division Algorithm for Unsigned Integer

Restoring division is usually performed on the fixed point fractional numbers. When we perform division operations on two numbers, the division algorithm will give us two things, i.e., quotient and remainder. This algorithm is based on the assumption that $0 < D < N$. With the help of digit set {0, 1}, the quotient digit q will be formed in the restoring division algorithm. The division algorithm is generally of two types, i.e., fast algorithm and slow algorithm. Goldschmidt and Newton-Raphson are the types of fast division algorithm, and STR algorithm, restoring algorithm, non-performing algorithm, and the non-restoring algorithm are the types of slow division algorithm.

In this section, we are going to perform restoring algorithm with the help of an unsigned integer. We are using restoring term because we know that the value of register A will be restored after each iteration. We will also try to solve this problem using the flow chart and apply bit operations.



Here, register Q is used to contain the quotient, and register A is used to contain the remainder. Here, the divisor will be loaded into the register M, and n-bit divided will be loaded into the register Q. 0 is the starting value of a register. The values of these types of registers are restored at the time of iteration. That's why it is known as restoring.



Now we will learn some steps of restoring division algorithm, which is described as follows:

Step 1: In this step, the corresponding value will be initialized to the registers, i.e., register A will contain value 0, register M will contain Divisor, register Q will contain Dividend, and N is used to specify the number of bits in dividend.

Step 2: In this step, register A and register Q will be treated as a single unit, and the value of both the registers will be shifted left.

Step 3: After that, the value of register M will be subtracted from register A. The result of subtraction will be stored in register A.

Step 4: Now, check the most significant bit of register A. If this bit of register A is 0, then the least significant bit of register Q will be set with a value 1. If the most significant bit of A is 1, then the least significant bit of register Q will be set to with value 0, and restore the value of A that means it will restore the value of register A before subtraction with M.

Step 5: After that, the value of N will be decremented. Here n is used as a counter.

Step 6: Now, if the value of N is 0, we will break the loop. Otherwise, we have to again go to step 2.

Step 7: This is the last step. In this step, the quotient is contained in the register Q, and the remainder is contained in register A.

For example:

In this example, we will perform a Division restoring algorithm.

1. Dividend = 11
2. Divisor = 3

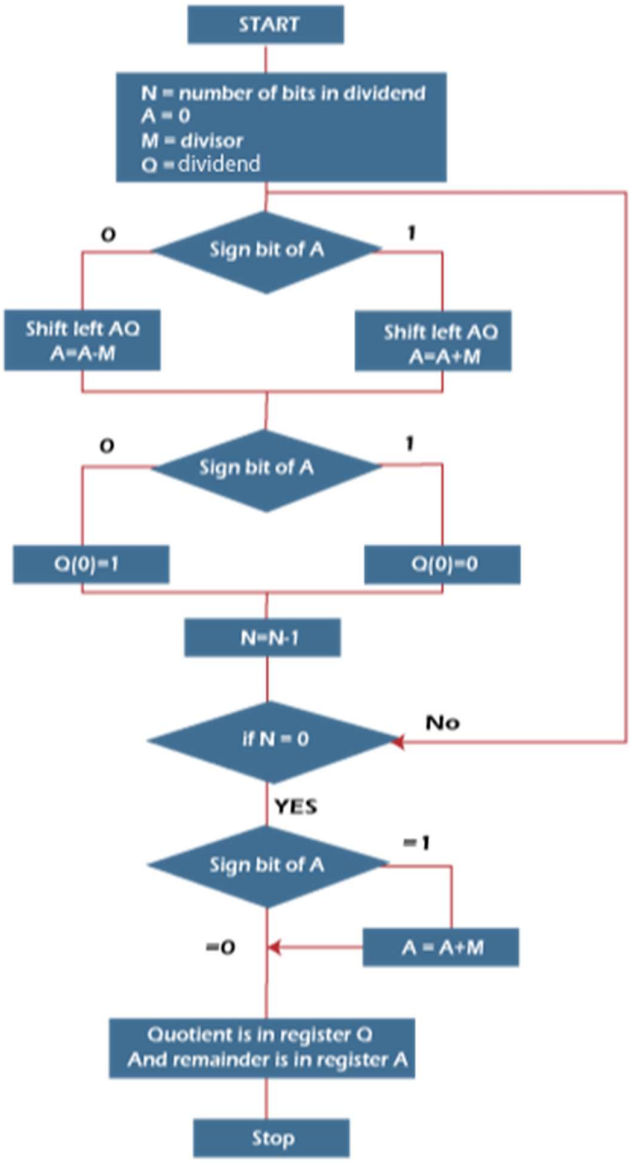
N	M	A	Q	Operation
4	00011	00000	1011	Initialize
	00011	00001	011_	Shift left AQ
	00011	11110	011_	A = A - M
	00011	00001	0110	Q[0] = 0 And restore A
3	00011	00010	110_	Shift left AQ
	0001 1	11111	110_	A = A - M
	00011	00010	1100	Q[0] = 0
2	00011	00101	100_	Shift left AQ
	00011	00010	100_	A = A - M
	00011	00010	1001	Q[0] = 1

1	00011	00101	001_	Shift left AQ
	00011	00010	001_	A = A - M
	00011	00010	0011	Q[0] = 1

So we should not forget to restore the value of the most significant bit of A, which is 1. So, register A contains the remainder 2, and register Q contains the quotient 3.

Non-Restoring Division Algorithm for Unsigned Integer

Instead of the quotient digit set {0, 1}, the set {-1, 1} is used by the non-restoring division. The non-restoring division algorithm is more complex as compared to the restoring division algorithm. But when we implement this algorithm in hardware, it has an advantage, i.e., it contains only one decision and addition/subtraction per quotient bit. After performing the subtraction operation, there will not be any restoring steps. Due to this, the numbers of operations basically cut down up to half. Because of the less operation, the execution of this algorithm will be fast. This algorithm basically performs simple operations such as addition, subtraction. In this method, we will use the sign bit of register A. 0 is the starting value/bit of register A.



Now we will learn steps of the non-restoring division algorithm, which are described as follows:

Step 1: In this step, the corresponding value will be initialized to the registers, i.e., register A will contain value 0, register M will contain Divisor, register Q will contain Dividend, and N is used to specify the number of bits in dividend.

Step 2: In this step, we will check the sign bit of A.

Step 3: If this bit of register A is 1, then shift the value of AQ through left, and perform A = A + M. If this bit is 0, then shift the value of AQ into left and perform A = A - M. That means in case of 0, the 2's complement of M is added into register A, and the result is stored into A.

Step 4: Now, we will check the sign bit of A again.

Step 5: If this bit of register A is 1, then Q[0] will become 0. If this bit is 0, then Q[0] will become 1. Here Q[0] indicates the least significant bit of Q.

Step 6: After that, the value of N will be decremented. Here N is used as a counter.

Step 7: If the value of N = 0, then we will go to the next step. Otherwise, we have to again go to step 2.

Step 8: We will perform A = A + M if the sign bit of register A is 1.

Step 9: This is the last step. In this step, register A contains the remainder, and register Q contains the quotient.

For example: In this example, we will perform a Non-Restoring Division algorithm with the help of an Unsigned integer.

1. Dividend = 11
2. Divisor = 3
3. -M = 11101

N	M	A	Q	Action
4	00011	00000	1011	Begin
	00011	00001	011_	Shift left AQ
	00011	11110	011_	A = A - M
3	00011	11110	0110	Q[0] = 0
	00011	11100	110_	Shift left AQ
	00011	11111	110_	A = A + M
2	00011	11111	1100	Q[0] = 0
	00011	11111	100_	Shift left AQ
	00011	00010	100_	A = A +M
1	00011	00010	1001	Q[0] = 1
	00011	00101	001_	Shift left AQ
	00011	00010	001_	A = A - M
0	00011	00010	0011	Q[0] = 1

So, register A contains the remainder 2, and register Q contains the quotient 3.

Floating Point Addition and Subtraction:

Compared to a fixed point addition and subtraction, a floating point addition and subtraction is more complex and hardware consuming. This is because exponent field is not present in case of fixed point arithmetic.

The major steps for a floating point addition and subtraction are

- Extract the sign of the result from the two sign bits.
- Subtract the two exponents and . Find the absolute value of the exponent difference ( ) and choose the exponent of the greater number.
- Shift the mantissa of the lesser number by bits Considering the hidden bits.
- Execute addition or subtraction operation between the shifted version of the mantissa and the mantissa of the other number. Consider the hidden bits also.

- **Normalization for addition:** In case of addition, if there is an carry generated then the result right shifted by 1-bit. This shift operation is reflected on exponent computation by an increment operation.

- **Normalization for subtraction:** A normalization step is performed if there are leading zeros in case of subtraction operation. Depending on the leading zero count the obtained result is left shifted. Accordingly the exponent value is also decremented by the number of bits equal to the number of leading zeros.

UNIT – III

8086 Architecture:

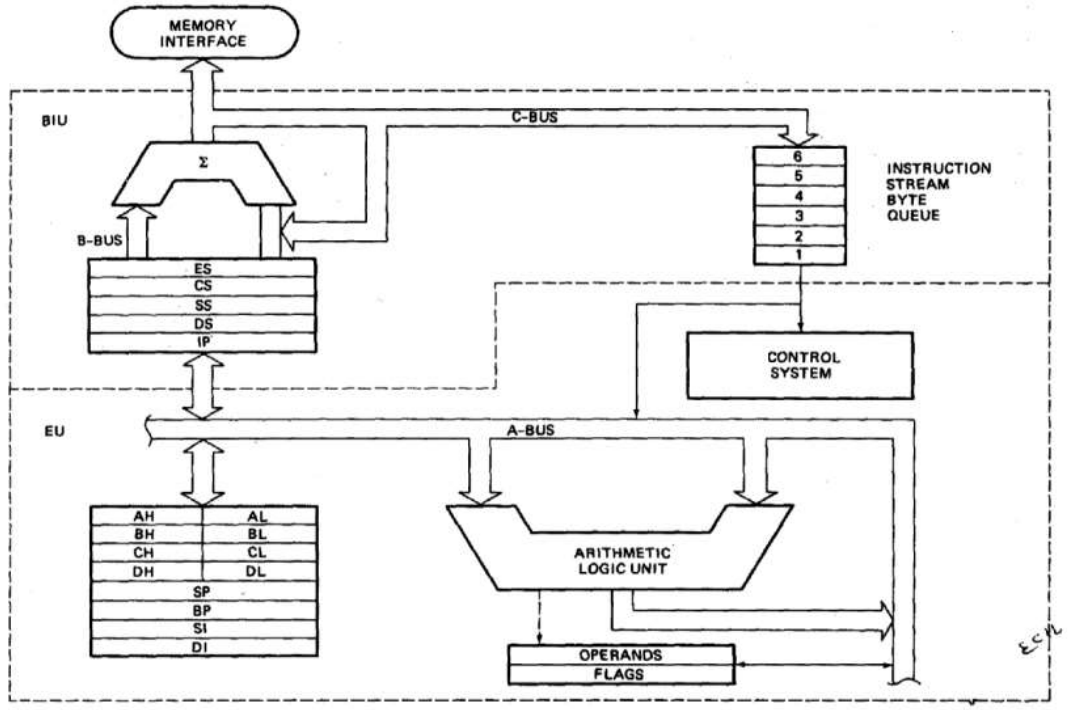


FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

8086 Internal block diagram

• 8086 has two blocks BIU (BUS INTERFACR UNIT) and EU (EXECUTION UNIT)

• The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.

• EU executes instructions from the instruction system byte queue.

• Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance.

• BIU contains Instruction queue, Segment registers, Instruction pointer, and Address adder.

• EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

1. BUS INTERFACR UNIT:

• It provides a full 16 bit bidirectional data bus and 20 bit address bus.

•The bus interface unit is responsible for performing all external bus operations

•Instruction fetch, Instruction queuing, Operand fetch and storage, Address relocation and Bus control.

•The BIU uses a mechanism known as an instruction stream queue to implement a pipeline architecture.

•This queue permits prefetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.

•These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle

•The BIU is also responsible for Generating bus control signals such as those for memory read or write and I/O read or write.

2. EXECUTION UNIT

 The Execution unit is responsible for decoding and executing all instructions.

 The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bys cycles to memory or I/O and perform the operation specified by the instruction on the operands.

 During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.

 If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.

 When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.

 Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

Design of Control Unit

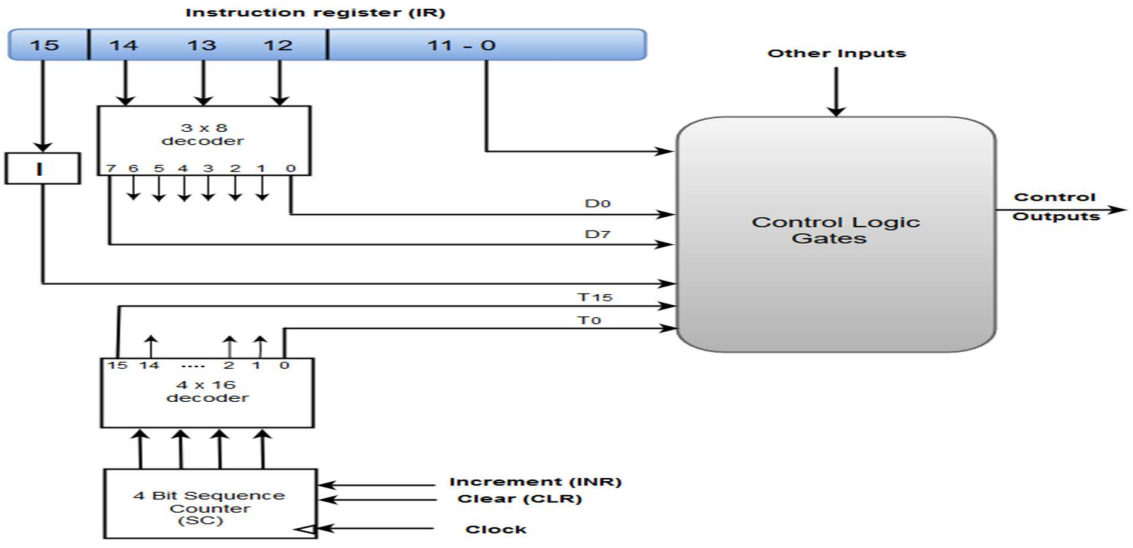
The Control Unit is classified into two major categories:

1. Hardwired Control
2. Microprogrammed Control

Hardwired Control

The Hardwired Control organization involves the control logic to be implemented with gates, flip-flops, decoders, and other digital circuits.The following image shows the block diagram of a Hardwired Control organization.

Control Unit of a Basic Computer:



- A Hard-wired Control consists of two decoders, a sequence counter, and a number of logic gates.
- An instruction fetched from the memory unit is placed in the instruction register (IR).
- The component of an instruction register includes; I bit, the operation code, and bits 0 through 11.

- The operation code in bits 12 through 14 are coded with a 3 x 8 decoder.
- The outputs of the decoder are designated by the symbols D0 through D7.
- The operation code at bit 15 is transferred to a flip-flop designated by the symbol I.
- The operation codes from Bits 0 through 11 are applied to the control logic gates.
- The Sequence counter (SC) can count in binary from 0 through 15.

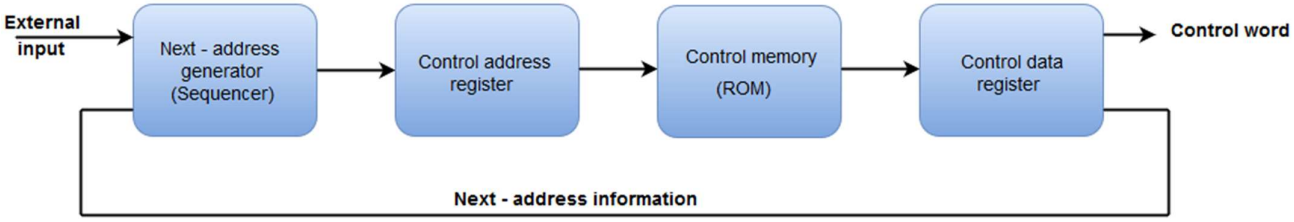
Micro-programmed Control

The Microprogrammed Control organization is implemented by using the programming approach.

In Microprogrammed Control, the micro-operations are performed by executing a program consisting of micro-instructions.

The following image shows the block diagram of a Microprogrammed Control organization.

Microprogrammed Control Unit of a Basic Computer:

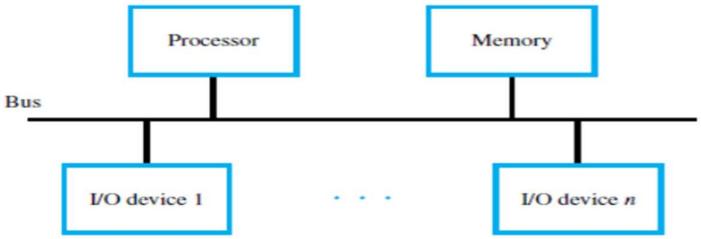


- The Control memory address register specifies the address of the micro-instruction.
- The Control memory is assumed to be a ROM, within which all control information is permanently stored.
- The control register holds the microinstruction fetched from the memory.
- The micro-instruction contains a control word that specifies one or more micro-operations for the data processor.
- While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- The next address generator is often referred to as a micro-program sequencer, as it determines the address sequence that is read from control memory.

Input-output subsystems:

The Input/output organization of computer depends upon the size of computer and the peripherals connected to it. The I/O Subsystem of the computer provides an efficient mode of communication between the central system and the outside environment. The most common input output devices are: Monitor, Keyboard, Mouse, Printer, Magnetic tapes Input Output Interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The

purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.



The Major Differences are:-

- Peripherals are electromechnical and electromagnetic devices and CPU and memory are electronic devices. Therefore, a conversion of signal values may be needed.
- The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a synchronization mechanism may be needed.
- Data codes and formats in the peripherals differ from the word format in the CPU and memory.
- The operating modes of peripherals are different from each other and must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and Peripherals to supervises and synchronizes all input and out transfers.

These components are called Interface Units because they interface between the processor bus and the peripheral devices. I/O device interface The I/O Bus consists of data lines, address lines and control lines. The I/O bus from the processor is attached to all peripherals interface.

To communicate with a particular device, the processor places a device address on address lines. Each Interface decodes the address and control received from the I/O bus, interprets them for peripherals and provides signals for the peripheral controller.

It is also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller.

For example, the printer controller controls the paper motion, the print timing. The control lines are referred as I/O command. The commands are as following:

Control command- A control command is issued to activate the peripheral and to inform it what to do.

Status command- A status command is used to test various status conditions in the interface and the peripheral.

Data Output command- A data output command causes the interface to respond by transferring data from the bus into one of its registers.

Data Input command- The data input command is the opposite of the data output.

In this case the interface receives on item of data from the peripheral and places it in its buffer register.

I/O Versus Memory Bus

To communicate with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address and read/write control lines.

There are 3 ways that computer buses can be used to communicate with memory and I/O:

1. Use two Separate buses, one for memory and other for I/O.

2. Use one common bus for both memory and I/O but separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

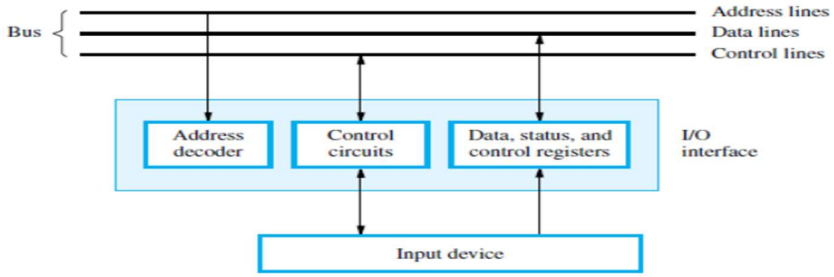


Figure 7.2 I/O interface for an input device.

MODES OF TRANSFER

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit.

Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as An intermediate path; other transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O

2. Interrupt-initiated I/O

3. Direct memory access (DMA)

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral.

Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CU can proceed to execute another program.

The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus.

When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMPA and interrupt facility.

In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.

EXAMPLE OF PROGRAMMED I/O

In the programmed I/O method, the I/O device dies not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory.

Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in below Fig. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line.

The interface sets it in the status register that we will refer to as an F or “flag” bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig.

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of The flag bit. If the flag is equal to 1, the CPU reads the data fromthe data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed.

Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. Flowcharts for CPU program to input data. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

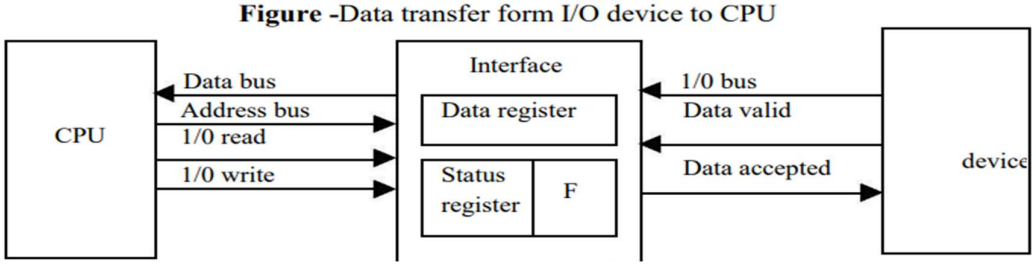
1. Read the status register.

2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.

3. Read the data register. Each byte is read into a CPU register and then transferred to memory with a store instruction.

A common I/O programming task is to transfer a block of words form an I/O device and store them in a memory buffer. A program that stores input characters in a memory buffer using the instructions mentioned in the earlier chapter.

Figure -Data transfer form I/O device to CPU



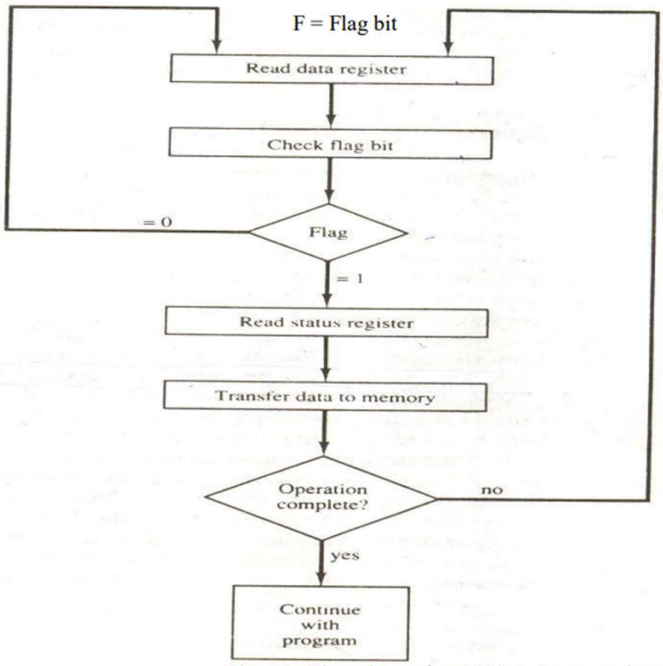


Figure -Flowcharts for CPU program to input data

The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.

To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 πs. Assume that the input device transfers its data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 π s. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

INTERRUPT-INITIATED I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag.

However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.

The way that the processor chooses the branch address of the service routine varies from tone unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non-vectored interrupt,

the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

DIRECT MEMORY ACCESS (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure- CPU bus signals for DMA transfer. Shows two control signals in the CPU that facilitate the DMA transfer.

The bus request (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit,

which means that the output is disconnected and does not have a logic significance. The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high -impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention.

When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA burst transfer, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called cycle stealing allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to “steal” one memory cycle.

DMA TRANSFER

The position of the DMA controller among the other components in a computer system is illustrated in below Fig. The CPU communicates with the DMA through the address and data buses as with any interface unit.

The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled.

The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. Note that the RD and WR lines in the DMA controller are bidirectional.

The direction of transfer depends on the status of the BG line. When BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR and output lines from the DMA controller to the randomaccess memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data us (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

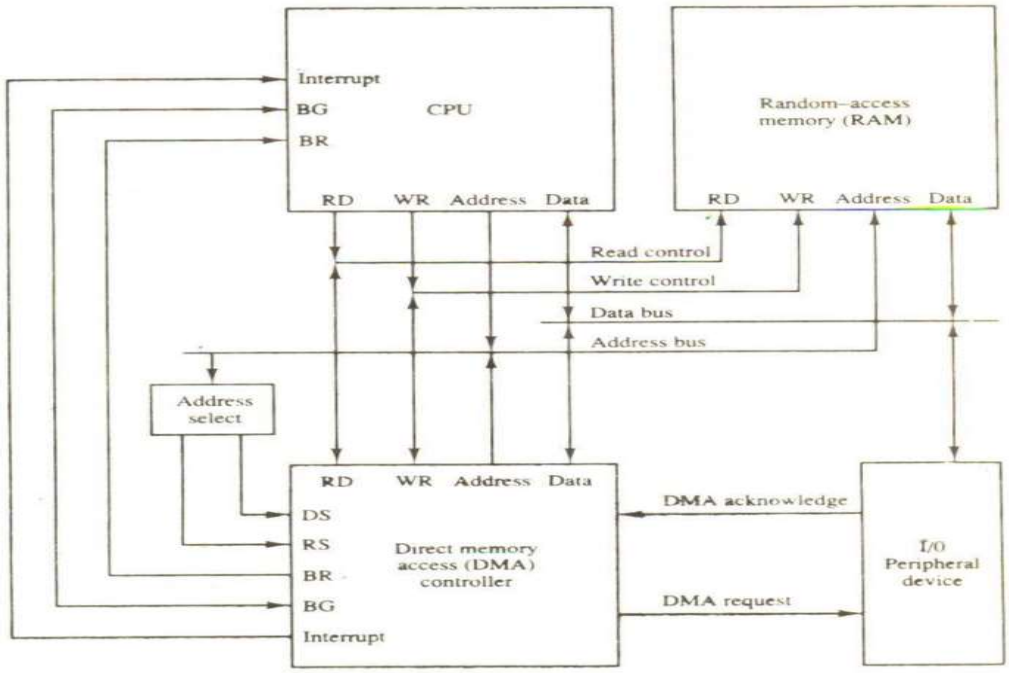


Figure-DMA transfer in a computer system.

For each word that is transferred, the DMA increments its address registers and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

It the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

A DMA controller may have more than on channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

Privileged Instructions and Non- Privileged Instructions:

Instructions are divided into two categories:

1)non-privileged instructions.

2)privileged instructions.

A non-privileged instruction is an instruction that any application or user can execute.

Examples of non-privileged instructions:

```
1 movl
2 addl
3 call
4 ret
```

A privileged instruction, on the other hand, is an instruction that can only be executed in kernel mode. Instructions are divided in this manner because privileged instructions could harm the kernel.

Examples of privileged instructions:

```
1 insl
2 outb
3 inb
4 int
```

Exceptions and Software interrupts:

Exceptions and interrupts are unexpected events that disrupt the normal flow of instruction execution. An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor. You are to implement exception and interrupt handling in your multicycle CPU design.

External interrupts come from input input (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction.

Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the

computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure.

A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

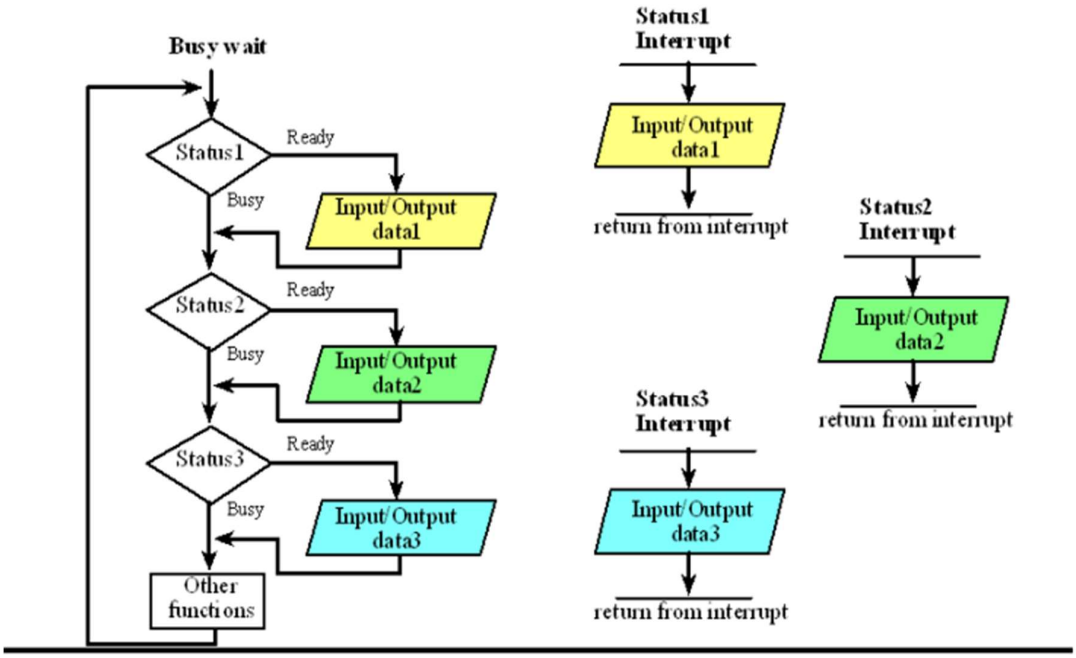
Programs and processes-Role of interrupts in process state transitions:

An interrupt is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a trigger. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer).

When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A thread is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a BX LR). A new thread is created for each interrupt request.

It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt. In a multi-threaded system, we consider the threads as cooperating to perform an overall task. Consequently we will develop ways for the threads to communicate (e.g., FIFO) and to synchronize with each other. Most embedded systems have a single common overall goal.

On the other hand, general-purpose computers can have multiple unrelated functions to perform. A process is also defined as the action of software as it executes. Processes do not necessarily cooperate towards a common shared goal. Threads share access to I/O devices, system resources, and global variables, while processes have separate global variables and system resources. Processes do not share I/O devices.



USB :

The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer.
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections.
- Enhance user convenience through a "plug-and-play" mode of operation.

USB Structure:

A serial transmission format has been chosen for the USB because a serial bus satisfies the lowcost and flexibility requirements.

Clock and data information are encoded together and transmitted as a single signal.Hence, there are no limitations on clock frequency or distance arising from data skew.

To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure. Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O device. At the root of the tree, a root hub connects the entire tree to the host computer.

The tree structure enables many devices to be connected while using only simple point-to-point serial links.

Each hub has a number of ports where devices may be connected, including other hubs.In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message.

A message sent from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

USB Protocols:

All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information.

The information transferred on the USB can be divided into two broad categories: control and data. ‹ Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error. Data packets carry information that is delivered to a device. For example, input and output data are transferred inside data packets

UNIT – IV

Basic concepts of pipelining: Performance of a computer can be increased by increasing the performance of the CPU. This can be done by executing more than one task at a time. This procedure is referred to as pipelining. The concept of pipelining is to allow the processing of a new task even though the processing of previous task has not ended.

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

Consider the following operation**: Result=(A+B)*C**

First the A and B values are Fetched which is nothing but a "Fetch Operation".

The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.

The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed. Finally the Result is again stored in the "Result" variable.

In this process we are using up-to 5 pipelines which are

Fetch Operation (A),

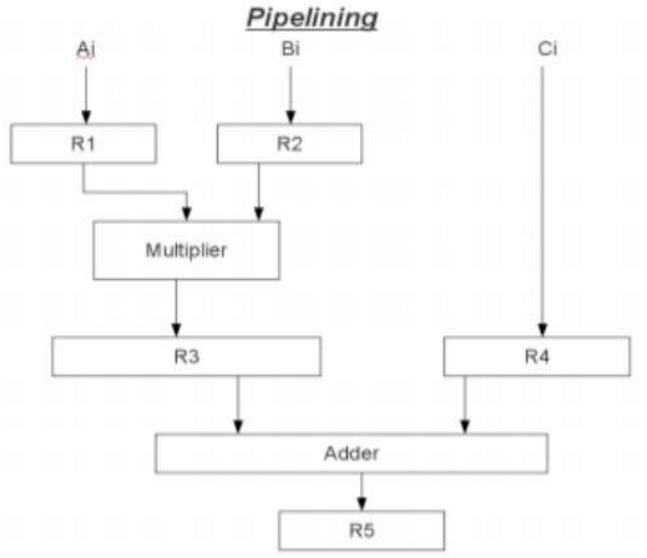
Fetch Operation(B),

Addition of (A & B),

Fetch Operation(C),

Multiplication of ((A+B), C),

Load ((A+B)*C).



Now consider the case where a k-segment pipeline with a clock cycle time t, is used to execute n tasks. The first task T1 requires a time equal to k t, to complete its operation since there are k segments in the pipe.

The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1)t, .

Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles.

For example, the diagram of Fig. shows four segments and six tasks.

The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the diagram.

TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Throughput and Speedup :

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system.

The purpose of parallel processing is to speed up the computer processing capability and increase its throughput.

Throughput: Is the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Speedup of a pipeline processing: The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = Tseq / Tpipe = n*m / (m+n -1)$$

the maximum speedup, also called ideal speedup, of a pipeline processor with m stages over an equivalent nonpipelined processor is m. In other words, the ideal speedup is equal to the number of pipeline stages. That is, when n is very large, a pipelined processor can produce output approximately m times faster than a nonpipelined processor. When n is small, the speedup decreases.

Pipeline Hazards:

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.

Hazards: The first hazard is called a structural hazard. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.

A **structural hazard** in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 4.27 had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard

Data Hazards:

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads must wait that have completed drying and are ready to fold as well as those that have finished washing and are ready to dry.

In a pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry).

For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

add $s0, $t0, $t1

sub $t2, $s0, $t3

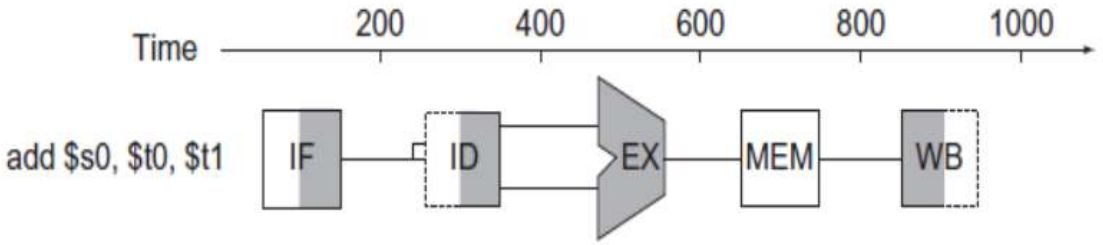


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline. Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory

access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.

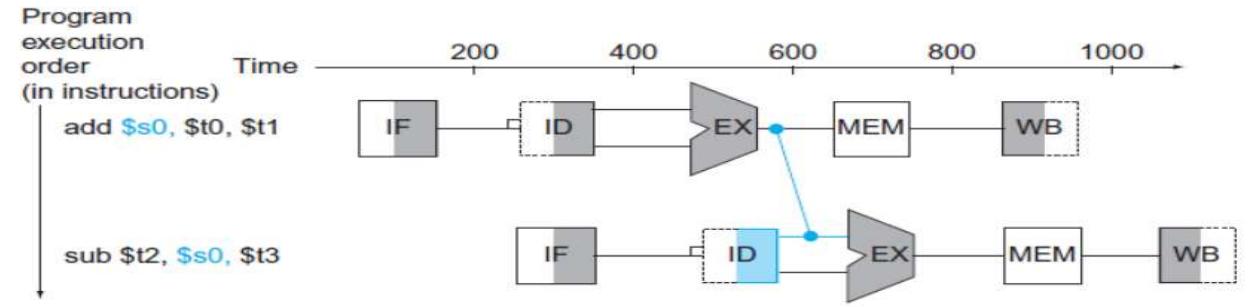


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path

It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of $s0 instead of an add. As we can imagine from looking at Figure 4.29, the desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a load-use data hazard, as Figure 4.30 shows. This figure shows an important pipeline concept, officially called a pipeline stall, but often given the nickname bubble. We shall see stalls elsewhere in the pipeline.

Control Hazards:

The third type of hazard is called a **control hazard**, arising from the need to make a decision based on the results of one instruction while others are executing. Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until aft er the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do? Here is the first of two solutions to control hazards in the laundry room and its computer equivalent.

Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula. This conservative option certainly works, but it is slow.

Parallel Processors

Introduction to parallel processors:

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of in a easing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques a.re economically feasible.

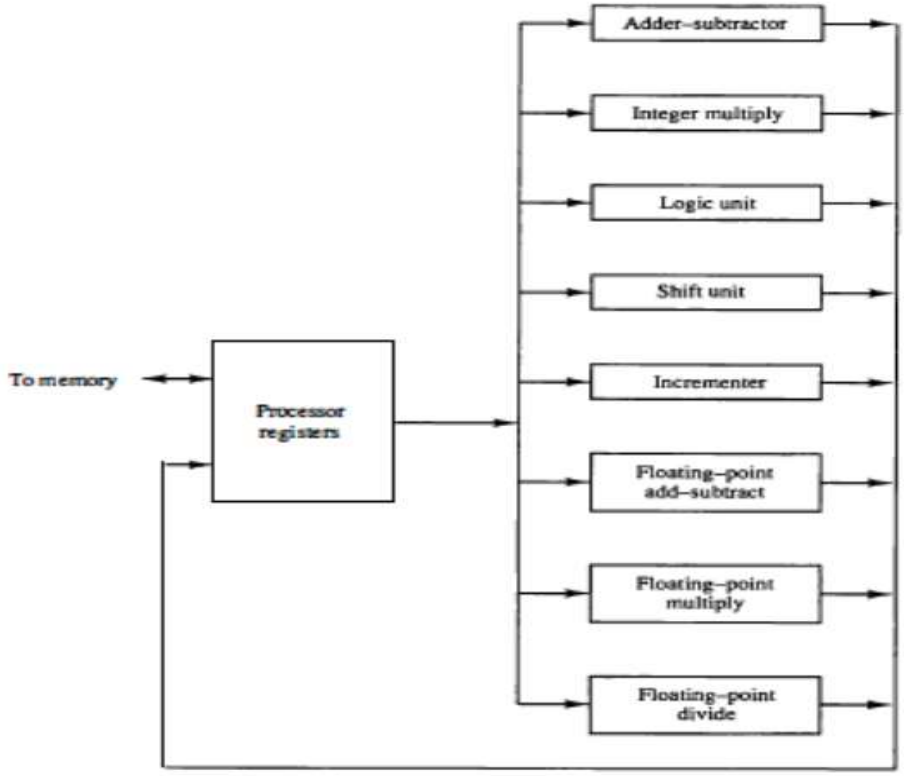
Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time, while registers with parallel load operate with all the bits of the word simultaneously.

Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift

operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers.

Figure 9-1 Processor with multiple functional units.



There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor.

The sequence of instructions read from memory constitutes an instruction stream . The operations performed on the data in the processor constitutes a data stream . Parallel processing may occur in the instruction stream, in the data stream, or in both.

Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction stream, single data stream (MISD)

Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.

Concurrent access to memory and cache coherency:

The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory.

Write-through policy: In the write-through policy, both cache and main memory are updated with every write operation.

Write-back policy: In the write-back policy, only the cache is updated and the location is marked so that it can be copied later into main memory.

In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory. To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical.

This requirement imposes a cache coherence problem. A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Without a proper solution to the cache coherence problem, caching cannot be used in busoriented multiprocessors with two or more processors.

Conditions for Incoherence:

Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data. Read-only data can safely be replicated without cache coherence enforcement mechanisms.

To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12. Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3. As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory. If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache.

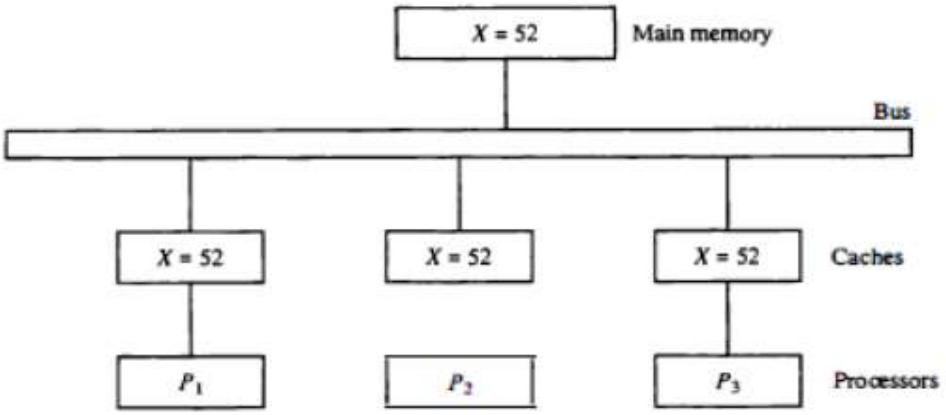
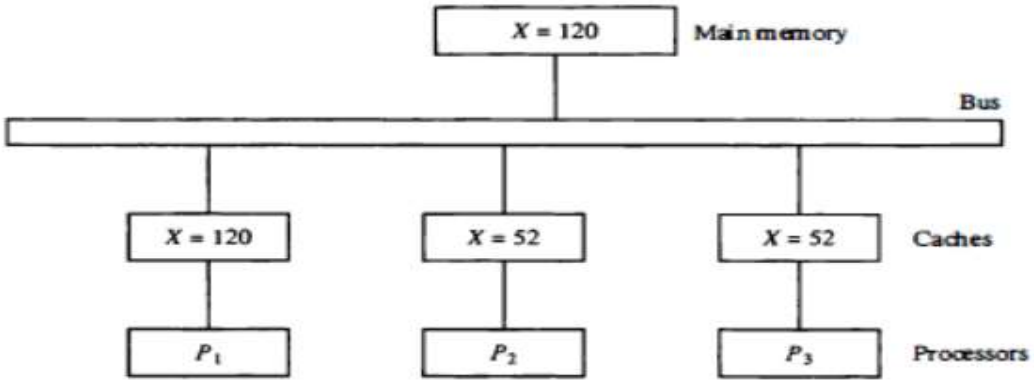


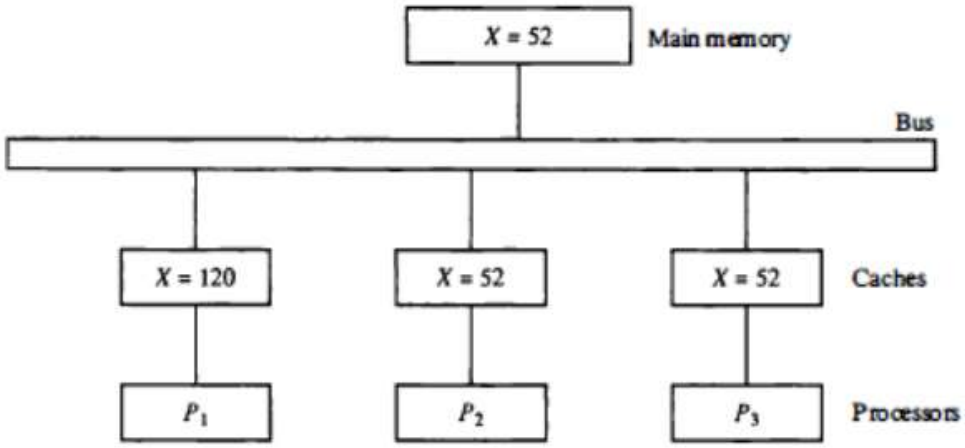
Figure 13-12 Cache configuration after a load on X.

This is shown in Fig. 13-13. A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value. In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Figure 13-13 Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy



(b) With write-back cache policy

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy. VO-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.

UNIT – V

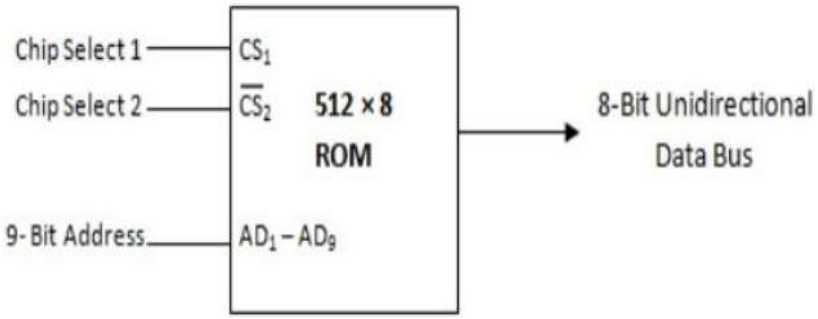
There are two main types or categories that can be used for semiconductor technology.

RAM - Random Access Memory: As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.

Block Diagram Representing 128 x 8 RAM
(Random Access Memory)

ROM - Read Only Memory: A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently,

even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.



The different memory types or memory technologies are detailed below:

DRAM: Dynamic RAM is a form of random access memory. DRAM uses a capacitor to store each bit of data, and the level of charge on each capacitor determines whether that bit is a logical 1 or 0. However these capacitors do not hold their charge indefinitely, and therefore the data needs to be refreshed periodically. As a result of this dynamic refreshing it gains its name of being a dynamic RAM. DRAM is the form of semiconductor memory that is often used in equipment including personal computers and workstations where it forms the main RAM for the computer.

EEPROM: This is an Electrically Erasable Programmable Read Only Memory. Data can be written to it and it can be erased using an electrical voltage. This is typically applied to an erase pin on the chip. Like other types of PROM, EEPROM retains the contents of the memory even when the power is turned off. Also like other types of ROM, EEPROM is not as fast as RAM.

EPROM: This is an Erasable Programmable Read Only Memory. This form of semiconductor memory can be programmed and then erased at a later time. This is normally achieved by exposing the silicon to ultraviolet light. To enable this to happen there is a circular window in the package of the EPROM to enable the light to reach the silicon of the chip. When the PROM is in use, this window is normally covered by a label, especially when the data may need to be preserved for an extended period. The PROM stores its data as a charge on a capacitor. There is a charge storage capacitor for each cell and this can be read repeatedly as required. However it is found that after many years the charge may leak away and the data may be lost. Nevertheless, this type of semiconductor memory used to be widely used in applications where a form of ROM was required, but where the data needed to be changed periodically, as in a development environment, or where quantities were low.

FLASH MEMORY: Flash memory may be considered as a development of EEPROM technology. Data can be written to it and it can be erased, although only in blocks, but data can be read on an individual cell basis. To erase and re-programme areas of the chip, programming voltages at levels that are available within electronic equipment are used. It is also non-volatile, and this makes it particularly useful. As a result Flash memory is widely used in many applications including memory cards for digital cameras, mobile phones, computer memory sticks and many other applications.

F-RAM: Ferroelectric RAM is a random-access memory technology that has many similarities to the standard DRAM technology. The major difference is that it incorporates a ferroelectric layer instead of the more usual dielectric layer and this provides its non-volatile capability. As it offers a non-volatile capability, F-RAM is a direct competitor to Flash.

MRAM: This is Magneto-resistive RAM, or Magnetic RAM. It is a non-volatile RAM memory technology that uses magnetic charges to store data instead of electric charges. Unlike technologies including DRAM, which require a constant flow of electricity to maintain the integrity of the data, MRAM retains data even when the power is removed. An additional advantage is that it only requires low power for active operation.

As a result this technology could become a major player in the electronics industry now that production processes have been developed to enable it to be produced.

P-RAM / PCM: This type of semiconductor memory is known as Phase change Random Access Memory, P-RAM or just Phase Change memory, PCM. It is based around a phenomenon where a form of chalcogenide glass changes is state or phase between an amorphous state (high resistance) and a polycrystalline state (low resistance). It is possible to detect the state of an individual cell and hence use this for data storage. Currently this type of memory has not been widely commercialized, but it is expected to be a competitor for flash memory.

PROM: This stands for Programmable Read Only Memory. It is a semiconductor memory which can only have data written to it once - the data written to it is permanent. These memories are bought in a blank format and they are programmed using a special PROM programmer. Typically a PROM will consist of an array of fuseable links some of which are "blown" during the programming process to provide the required data pattern.

SDRAM: Synchronous DRAM. This form of semiconductor memory can run at faster speeds than conventional DRAM. It is synchronised to the clock of the processor and is capable of keeping two sets of memory addresses open simultaneously. By transferring data alternately from one set of addresses, and then the other, SDRAM cuts down on the delays associated with non-synchronous RAM, which must close one address bank before opening the next.

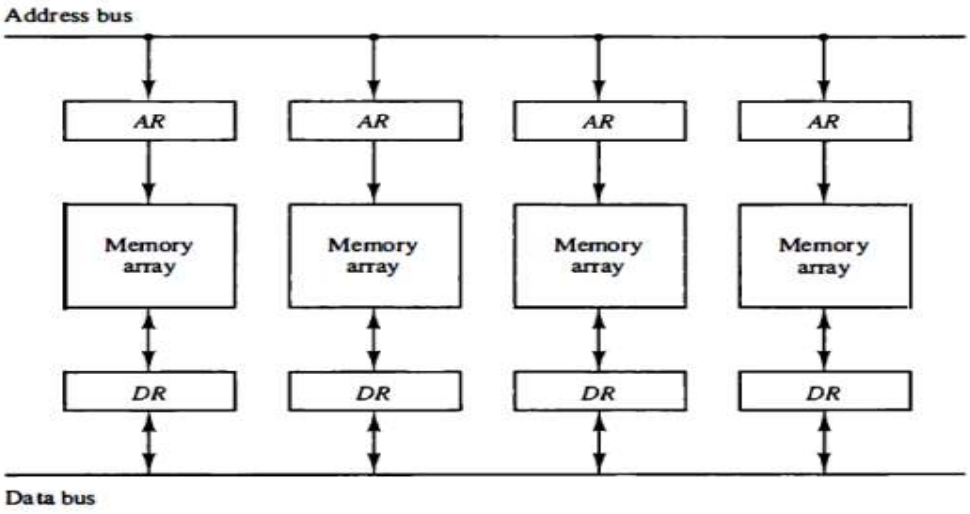
SRAM: Static Random Access Memory. This form of semiconductor memory gains its name from the fact that, unlike DRAM, the data does not need to be refreshed dynamically. It is able to support faster read and write times than DRAM (typically 10 ns against 60 ns for DRAM), and in addition its cycle time is much shorter because it does not need to pause between accesses. However it consumes more power, is less dense and more expensive than DRAM. As a result of this it is normally used for caches, while DRAM is used as the main semiconductor memory technology.

MEMORY ORGANIZATION:

Memory Interleaving: Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.

Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows a memory unit with four modules. Each memory array has its own address register AR and data register DR.

Figure 9-13 Multiple module memory organization.



The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

The advantage of a modular memory is that it allows the use of a technique called interleaving. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

Concept of Hierarchical Memory Organization:

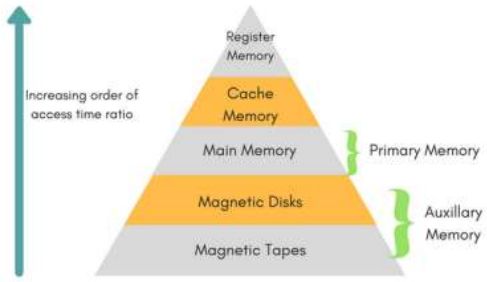
This Memory Hierarchy Design is divided into 2 main types:

External Memory or Secondary Memory

Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.

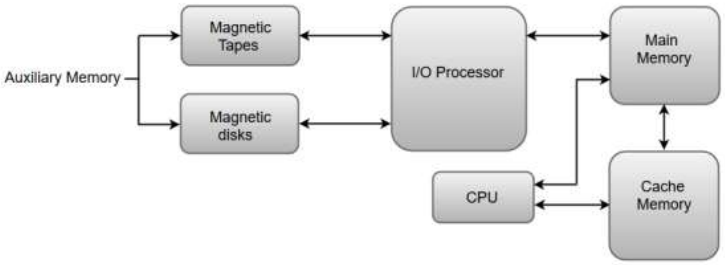
Internal Memory or Primary Memory

Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.



Characteristics of Memory Hierarchy

Memory Hierarchy in a Computer System:



Cache Memories:

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.

Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.

The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's replacement algorithm.

Cache Hits: The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.

If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a read or write hit is said to have occurred.

Cache Misses: A Read operation for a word that is not in the cache constitutes a Read miss. It causes the block of words containing the requested word to be copied from the main memory into the cache.

Cache Mapping: There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.

Direct mapping:The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on.

Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.

For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address.

The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block.

When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored.

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 first be read from the main memory and loaded into the cache.

The direct-mapping technique is easy to implement, but it is not very flexible.

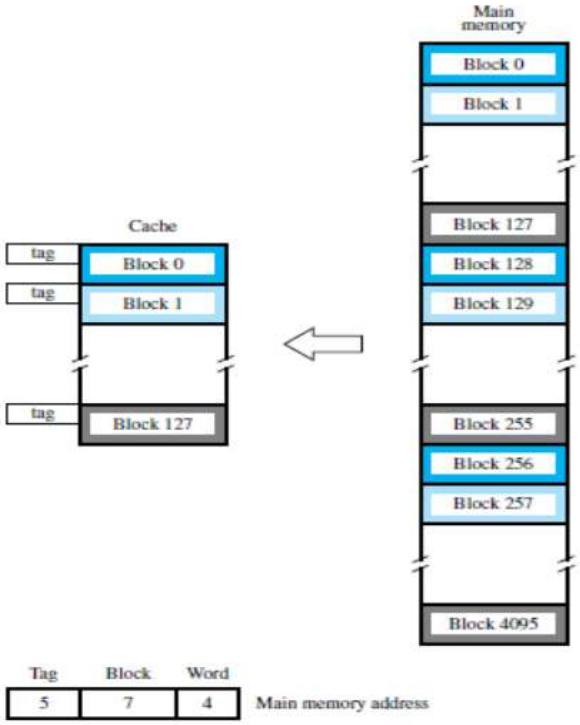


Figure 8.16 Direct-mapped cache.

Associative Mapping:

In Associative mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident 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 the associative-mapping technique.

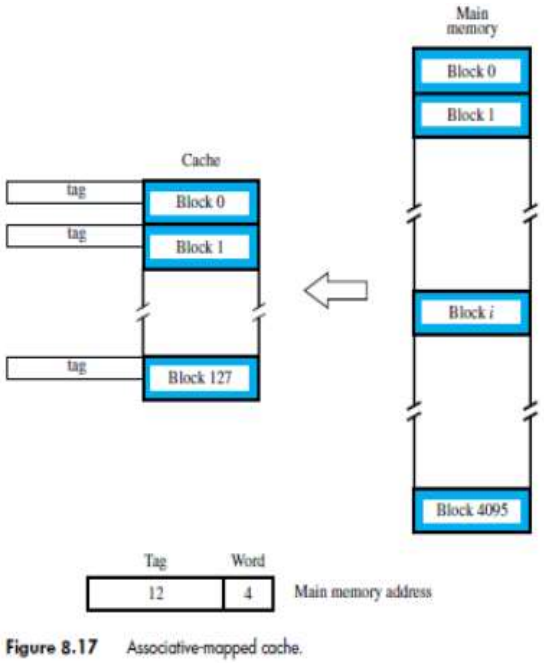


Figure 8.17 Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced.

To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an associative search.

Set-Associative Mapping:

Another approach is to use a combination of the direct- and associative-mapping techniques. 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 a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.

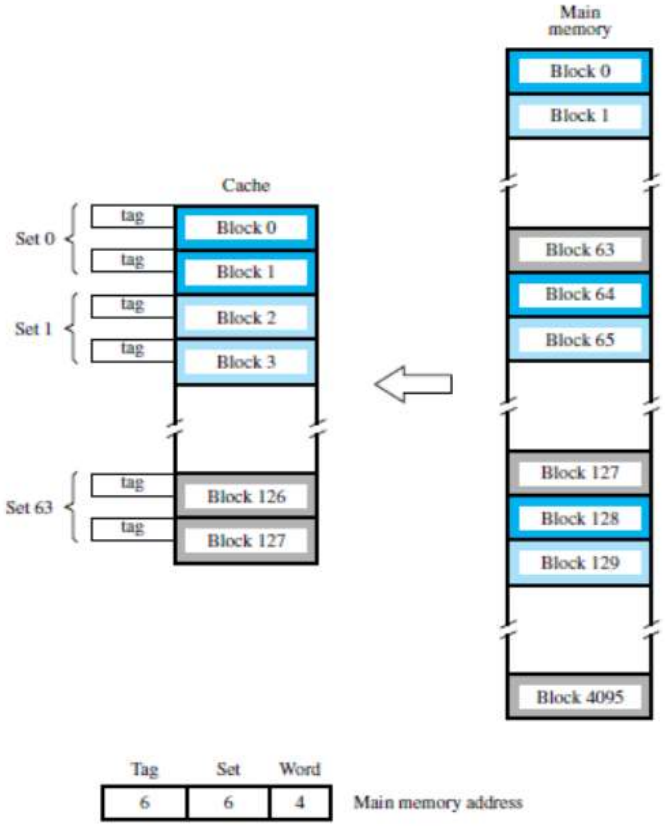


Figure 8.18 Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this set-associative-mapping technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set.

Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. 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. But, 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 program execution usually stays 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 (LRU) block, and the technique is called the LRU replacement algorithm.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases.

Write Policies:

The write operation is proceeding in 2 ways.

- 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.
- To overcome the read miss Load –through / Early restart protocol is used.