COMPUTER ORGANIZATION AND ARCHITECTURE

Course code: ACSB07

IV. B.Tech II semester

Regulation: IARE R-18

Prepared by:

Mr. E Sunil Reddy, Assistant Professor
Ms. B.Dhana Laxmi, Assistant Professor
Dr.P.L.Srinivasa Murthy. Professor
Mr. N Rajasekhar, Assistant Professor

COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500 043

Course Outcomes

The course should enable the students to:	
CO 1	Understand the organization and levels of design in computer architecture and To understand the concepts of programming methodologies.
CO 2	Describe Register transfer languages, arithmetic micro operations, logic micro operations, shift micro operations address sequencing, micro program example, and design of control unit.
CO 3	Understand the Instruction cycle, data representation, memory reference instructions, input-output, and interrupt, addressing modes, data transfer and manipulation, program control. Computer arithmetic: Addition and subtraction, floating point arithmetic operations, decimal arithmetic unit.
CO 4	Knowledge about Memory hierarchy, main memory, auxiliary memory, associative memory, cache memory, virtual memory Input or output Interface, asynchronous data transfer, modes of transfer, priority interrupt, direct memory access.
CO 5	Explore the Parallel processing, pipelining-arithmetic pipeline, instruction pipeline Characteristics of multiprocessors, inter connection structures, inter processor arbitration, inter processor Communication and synchronization

MODULE –I

INTRODUCTION TO COMPUTER ORGANIZATION

Course Learning Outcomes

The course will enable the students to:	
CLO 1	Describe the various components like input/output units, memory unit, control unit, arithmetic logic unit connected in the basic organization of a computer.
CLO 2	Understand the interfacing concept with memory subsystem organization and input/output subsystem organization.
CLO 3	Understand instruction types, addressing modes and their formats in the assembly language programs.
CLO 4	Describe the instruction set architecture design for relatively simple microprocessor or Central Processing Unit.
CLO 5	Understand the organization and levels of design in computer architecture and To understand the concepts of programming methodologies.

Contents

Basic computer organization

- **CPU organization**
- **Memory subsystem organization and interfacing**
- **Input or output subsystem organization and interfacing**
- **A simple computer levels of programming languages**
- **Assembly language instructions**
- **Instruction set architecture design**
- **A simple instruction set architecture.**

Basic Computer Organization

- The basic computer organization has three main components:
- CPU
- Memory subsystem
- I/O subsystem

Generic computer Organization

CPU

Address Bus

Data Bus

Control Bus

Memory Subsystem

I/O Device ... I/O Device

I/O Subsystem

system bus

- **The system bus has three buses:**
- **Address bus:** The uppermost bus is the address bus. When the CPU reads data or instructions from or writes data to memory, it must specify the address of the memory location it wishes to access.
- **Data bus:** Data is transferred via the Data bus. When CPU fetches data from memory it first outputs the memory address on to its address bus. Then memory outputs the data onto the data bus. Memory then reads and stores the data at the proper locations.
- **Control bus:** Control bus carries the control signal. Control signal is the collection of individual control signals. These signals indicate whether data is to be read into or written out of the CPU.

Instruction Cycle

- Each program is a sequence of instructions.
- The basic computer system each instruction is subdivided into Four phases:

  1. Fetch an Instruction from memory.

  2. Decode the Instruction.

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

  4. Execute the Instruction.

- The above process continues indefinitely unless a HALT instruction is encountered.

Instruction Cycle

Fetch and Decode

- Initially the program counter PC is loaded with the address of the first instruction.

$T0: AR \leftarrow PC \; (S_0S_1S_2=010, T0=1)$

$T1: IR \leftarrow M[AR], \; PC \leftarrow PC + 1 \; (S0S1S2=111, T1=1)$

$T2: D0, \ldots, D7 \leftarrow Decode \; IR(12\text{-}14), AR \leftarrow IR(0\text{-}11), I \leftarrow IR(15)$

Instruction Cycle

T1
T0
S_2
S_1
S_0
Bus
Memory unit
7
Address
Read
AR
1
LD
PC
2
INR
IR
5
LD
Clock
Common bus

Fig: Register Transfer for the fetch phase

Instruction Cycle

- In the above diagram shows the transfer of first two statements(T0 and T1).
- When timing signal T0=1then

1. Place the contents of PC onto the bus by making the bus selection inputs S2S1S0 equal to 010.
2. Transfer the contents of the bus to AR by enabling the LD input of AR.

- When timing signal T1=1 then

1. Enable the read input of memory.
2. Place the contents of Memory onto the bus by making S2S1S0=111.
3. Transfer the contents of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

Determine the Type of Instruction

- After executing the timing signal T1 the control unit determines the type of instruction that is read from memory.
- If D7=1 and the instruction must be a register-reference or input-out type.
- If D7=0 the operation code must be one of the other seven values 000 through 110 specifying a memory –reference Instruction.
- The symbolic representation is :

D'7IT3 : AR ←M[AR]
D'7I'T3 : Nothing
D7I'T3 : Execute a register-reference instr.
D7IT3 : Execute an input-output instr.

Instruction Cycle

Start
SC ← 0

T0
AR ← PC

T1
IR ← M[AR], PC ← PC + 1

T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)

D7

(Register or I/O) = 1

= 0 (Memory-reference)

I

(I/O) = 1

= 0 (register)

I

(indirect) = 1

= 0 (direct)

T3
Execute
input-output
instruction
SC ← 0

T3
Execute
register-reference
instruction
SC ← 0

T3
AR ← M[AR]

T3
Nothing

T4
Execute
memory-reference
instruction
SC ← 0

Fig : Flowchart for Instruction Cycle

Control signals

- The **READ** signal is a signal on the control bus which the microprocessor asserts when it is ready to read data from memory or I/O device.
- When **READ** signal is asserted the memory subsystem places the instruction code be fetched on to the computer system's data bus. The microprocessor then inputs the data from the bus and stores its internal register.
- **READ** signal causes the memory to read the data, the **WRITE** operation causes the memory to store the data

Timing diagrams

clock cycle 1
clock cycle 2
CLK
Address bus
Address
Data bus
Data
READ

(a)

clock cycle 1
clock cycle 2
CLK
Address bus
Address
Data bus
Data
WRITE

(b)

Memory read operation

- In fig (a) the microprocessor places the address on to the bus at the beginning of a clock cycle, a 0/1 sequence of clock. One clock cycle later, to allow for memory to decode the address and access its data, the microprocessor asserts the READ control signal.
- This causes the memory to place its data onto the system data bus. During this clock cycle, the microprocessor reads the data off the system bus and stores it in one of the registers.
- At the end of the clock cycle it removes the address from the address bus and de asserts the READ signal. Memory then removes the data from the data from the data bus completing the memory read operation

Memory write operation

- In fig(b) the processor places the address and data onto the system bus during the first clock pulse.
- The microprocessor then asserts the WRITE control signal at the end of the second clock cycle.
- At the end of the second clock cycle the processor completes the memory write operation by removing the address and data from the system bus and deasserting the WRITE signal.

CPU Organization

- Central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.
- In the computer all the all the major connected with the help of the **system bus**.
- **Data bus** is used to shuffle data between the various components in a computer system.
- When the software wants to access some particular memory location or I/O device it places the corresponding address on the **address bus**.
- The **control bus** is an eclectic collection of signals that control how the processor communicates with the rest of the system. The **read** and **write** control lines control the direction of data on the data bus.

CPU Organization

The Microprocessor

Control unit

Arithmetic-logic unit

Registers

Primary storage (main memory)

Input

Output

Communication devices

Secondary storage

CPU Organization

- The register section, as its name implies, includes a set of registers and a bus or other communication mechanism.
- The register in a processor's instruction set architecture are found in the section of the CPU.
- The system address and data buses interact with this section of CPU. The register section also contains other registers that are not directly accessible by the programmer.
- The fetch portion of the instruction cycle, the processor first outputs the address of the instruction onto the address bus. The processor has a register called the **program counter**.
- At the end of the instruction fetch, the CPU reads the instruction code from the system data bus.
- It stores this value in an internal register, usually called the **instruction register".**

CPU Organization

- The **arithmetic / logic unit (or) ALU** performs most arithmetic and logic operations such as adding and ANDing values.
- CPU controls the computer, the **control unit** controls the CPU. The control unit receives some data values from the register unit, which it used to generate the control signals.
- The **control unit** also generates the signals for the system control bus such as READ, WRITE, IO/ signals

Memory Subsystem Organization

- Memory is the group of circuits used to store data.
- Memory components have some number of memory locations, each word of which stores a binary value of some fixed length.
- The number of locations and the size of each location vary from memory chip to memory chip, but they are fixed within individual chip.
  - Memory is usually organized in the form of arrays, in which each cell is capable of storing one bit information.
  - Each row of cell constitutes a memory word, and all cells of a row are connected to a common column called word line, which is driven by the address decoder on the chip

Types of Memory

- There are **two types** of memory chips,

1. Read Only Memory (ROM)
2. Random Access Memory (RAM)

- Masked ROM(or) simply ROM
- PROM(Programmed Read Only Memory)
- EPROM(Electrically Programmed Read Only Memory)
- EEPROM(Electrically Erasable PROM)
- Flash Memory

ROM Chips

Masked ROM

- A masked ROM or simply ROM is programmed with data as chip is fabricated.
- The mask is used to create the chip and chip is designed with the required data hardwired in it.
- **PROM**
- Some ROM designs allow the data to be loaded by the user, thus providing programmable ROM (PROM).
- Once a program has been written onto a PROM, it remains there forever.
- Unlike main memory, PROMs retain their contents when the computer is turned off.
- PROM - (programmable read-only memory) is a memory chip on which data can be written only once.

ROM Chips

- **EPROM**
- EPROM is the another ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable reprogrammable ROM is usually called an EPROM.

- EPROM is done by charging of capacitors. The charged and uncharged capacitors cause each word of memory to store the correct value.
- The chip is erased by beingplaced under UV light,which causes the capacitor to leak their charge.

ROM Chips

- **EEPROM**
- A significant disadvantage of the EPROM is the chip is physically removed from the circuit for reprogramming and that entire contents are erased by the UV light.
- Another version of EPROM is EEPROM that can be both programmed and erased electrically, such chips called EEPROM, do not have to remove for erasure.
- The only disadvantage of EEPROM is that different voltages are need for erasing, writing, reading and stored data
- **Flash Memory**

- A special type of EEPROM is called a flash memory is electrically erase data in blocks rather than individual locations.
- It is well suited for the applications that writes blocks of data and can be used as a solid state hard disk. It is also used for data storage in digital computers.

Memory Subsystem Organization

- **RAM Chips:**
  - RAM stands for Random access memory. This often referred to as read/write memory. Unlike the ROM it initially contains no data.
  - The data pins are bidirectional unlike in ROM.
  - A ROM chip loses its data once power is removed so it is a volatile memory.
  - RAM chips are differentiated based on the data they maintain.

- Dynamic RAM (DRAM)
- Static RAM (SRAM)

Memory chips Internal organization

- **Dynamic RAM**
- DRAM chips are like leaky capacitors. Initially data is stored in the DRAM chip, charging its memory cells to their maximum values.
- The charging slowly leaks out and would eventually go too low to represent valid data.
- Before this a refresher circuit reads the content of the DRAM and rewrites data to its original locations.
- DRAM is used to construct the RAM in personal computers.
- **Static RAM**
- Static RAM are more likely the register .Once the data is written to SRAM, its contents stay valid it does not have to be refreshed.
- Static RAM is faster than DRAM but it is also much more expensive. Cache memory in the personal computer is constructed from SRAM.

Memory subsystem configuration

(b)

Multi byte organization

- There are two commonly used organizations for multi byte data.

  – Big endian
  – Little endian

- In BIG-ENDIAN systems the most significant byte of a multi- byte data item always has the lowest address, while the least significant byte has the highest address.

- In LITTLE-ENDIAN systems, the least significant byte of a multi-byte data item always has the lowest address, while the most significant byte has the highest address.

I/O Subsystem Organization

The I/O subsystem is treated as an independent unit in the computer The CPU initiates I/O commands generically

- Read, write, scan, etc.
- This simplifies the CPU

Input Device

The I/O subsystem is treated as an independent unit in the computer The CPU initiates I/O commands generically

- Read, write, scan, etc.
- This simplifies the CPU

Input device
Tri-state buffers
E
To data bus
Address bus
Control bus
Enable logic
(a)
RD
To buffer enable
IO/M
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
(b)

Input Device

- The data from the input device goes to the tri-state buffers. When the value in the address and control buses are correct, the buffers are enabled and data passes on the data bus.
- The CPU can then read this data. If the conditions are not right the logic block does not enable the buffers and do not place on the bus.
- The enable logic contains 8-bit address and also generates two control signals RD and I/O.

Output Device

- The design of the interface circuitry for an output device such as a computer monitor is somewhat different than for the input device.
- Tri-state buffers are replaced by a register.
- The tri-state buffers are used in input device interfaces to make sure that one device writes data to the bus at any time.
- Since the output devices read from the bus, rather that writes data to it, they don't need the buffers.
- The data can be made available to all output devices but the devices only contains the correct address will read it in

Output Device

An output device: (a) with its interface and (b) the enable logic for the registers

- Some devices are used for both input and output. Personal computer and hard disk devices are falls into this category. Such a devices requires a combined interface that is essential two interfaces.
- A bidirectional I/O device with its interface and enable load logic is shown in the Figure below.

Output Device

Tri-state buffers
E
To data bus
RD
I/O device
clk
LD
Register
From data bus
WR
IO/~M
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Figure : A bidirectional I/O device with its interface and enable/load logic

A Simple Computer- Levels of PL

- Computer programming languages are divided into 3 categories.
- High level language
- Assembly level language
- Machine level language
- **High level languages** are platform independent that is these programs can run on computers with different microprocessor and operating systems without modifications. Languages such as C++, Java and FORTRAN are high level languages.
- **Assembly languages** are at much lower level of abstraction. Each processor has its own assembly language
- The lowest level of programming language is **machine level** languages. These languages contain the binary values that cause the microprocessor to perform certain operations. When microprocessor reads and executes an instruction it's a machine language instruction.

A Simple Computer- Levels of PL

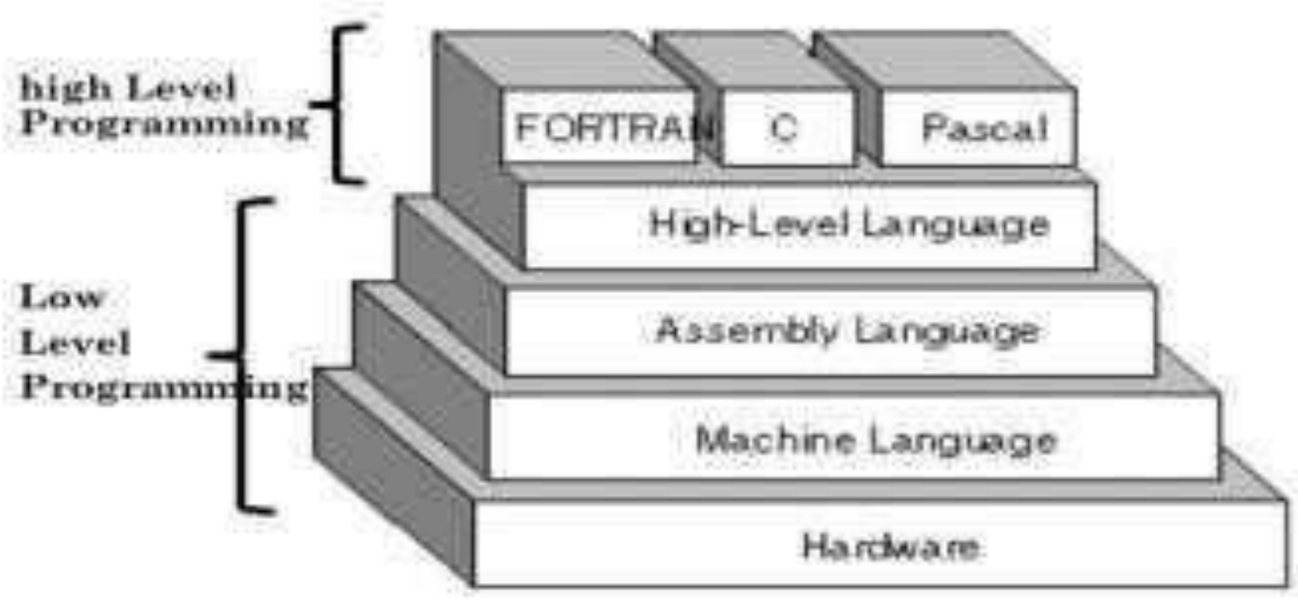


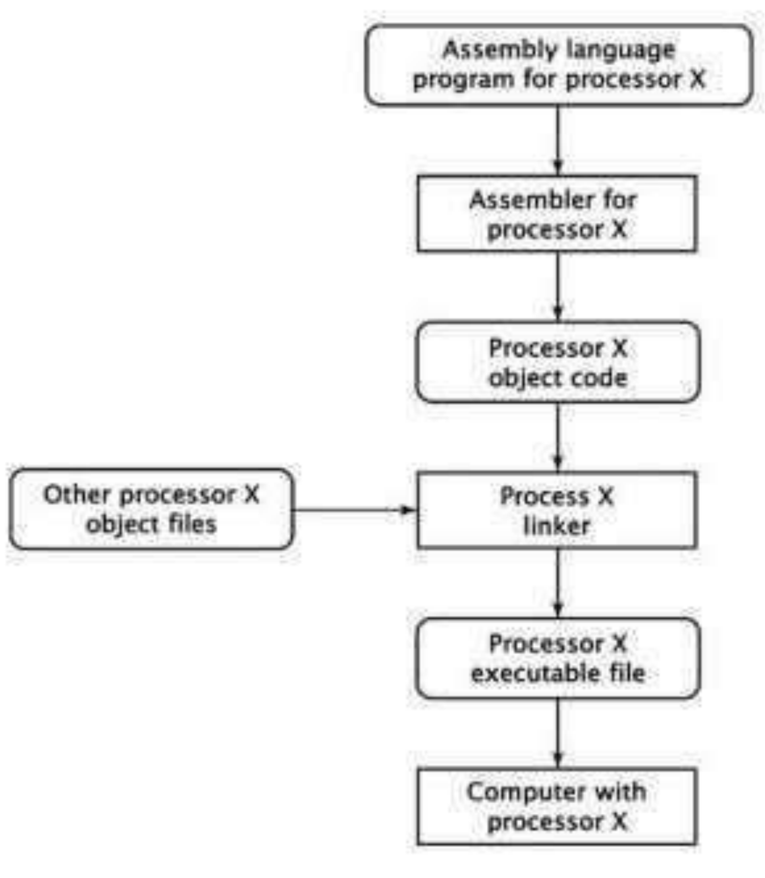
Figure 1.8: Levels of programming languages

A Simple Computer- Levels of PL

- High level language programs are compiled and assembly level language programs are assembled.
- A program written in the high level language is input to the compiler.
- compiler checks to make sure every statement in the program is valid. When the program has no syntax errors the compiler finishes the compiling the program that is source code and generates an object code file.
- An **object code** is the machine language equivalent of source code.
- A **linker** combines the object code to any other object code. This combined code stores in the **executable file**.

A Simple Computer- Levels of PL

Assembling programs



- Assembly language is specific to one microcontroller
- Converts the source code into object code
- The linker will combine the object code of your program with any other required object code to produce executable code
- Loader will load the executable code into memory, for execution

A Simple Computer- Levels of PL

- Programmers don't written the programs in machine language rather programs written in assembly or high level are the converted into machine level and then executed by microprocessor.
- High level language programs are compiled and assembly level language programs are assembled.
- A program written in the high level language is input to the compiler. The compiler checks to make sure every statement in the program is valid. When the program has no syntax errors the compiler finishes the compiling the program that is source code and generates an object code file.
- An **object code** is the machine language equivalent of source code.
- A **linker** combines the object code to any other object code. This combined code stores in the executable **file.**

Assembly Language Instructions

- **Instruction types**
- Assembly languages instructions are grouped together based on the operation they performed.
- Data transfer instructions
- Data operational instructions
- Program control instructions
- **Data transfer instructions**
- **Load the data from memory into the microprocessor**: These instructions copy data from memory into a microprocessor register.
- **Store the data from the microprocessor into the memory**: This is similar to the load data expect data is copied in the opposite direction from a microprocessor register to memory.
- **Move data within the microprocessor**: These operations copies data from one microprocessor register to another.
- **Input the data to the microprocessor**: The microprocessor inputs the data from the input devices ex: keyboard in to one of its registers.

Assembly Language Instructions

- **Data operational instructions**
- **Data operational instructions** do modify their data values. They typically perform some operations using one or two data values (operands) and store result.

- **Arithmetic instructions** make up a large part of data operations instructions. Instructions that add, subtract, multiply, or divide values fall into this category. An instruction that increment or decrement also falls in to this category.
- **Logical instructions** perform basic logical operations on data. They AND, OR, or XOR two data values or complement a single value.
- **Shift operations** as their name implies shift the bits of a data values also comes under this category

Assembly Language Instructions

- **Program control instructions**
- Program control instructions are used to control the flow of a program. Assembly language instructions may include subroutines like in high level language program may have subroutines, procedures, and functions.
- A jump or branch instructions are generally used to go to another part of the program or subroutine.

Instruction Set Architecture Design

- Designing of the instructions the most important in designing the microprocessor. A poor designed ISA even it is implemented well leads to bad micro processor.
- A well designed instruction set architecture on the other hand can result in a powerful processor that can meet variety of needs.
- In designing ISA the designer must evaluate the tradeoffs in performance and such constrains issues as size and cost when designing ISA specifications.
- The issue of **completeness** of the ISA is one of the criteria in designing the processor that means the processor must have complete set of instructions to perform the task of the given application.
- Another criterion is instruction **orthogonality.** Instructions are orthogonal if they do not overlap, or perform the same function. A good instruction set minimizes the overlap between instructions.

Instruction Set Architecture Design

- Another area that the designer can optimize the ISA is the **register set**. Registers have a large effect on the performance of a CPU. The CPU Can store data in its internal registers instead of memory. The CPU can retrieve data from its registers much more likely than form the memory.
- Having too few registers causes a program to make more reference to the memory thus reducing performance

Simple Instruction Set

- This processor inputting the data from and outputting the data to external devices such as microwave ovens keypad and display are treated as memory accesses. There are two types of input/output interactions that can design a CPU to perform.
- An **isolated I/O** input and output devices are treated as being separated from memory. Different instructions are used for memory and I/O.
- **Memory mapped I/O** treats input and output devices as memory locations the CPU access these I/O devices using the same instructions that it uses to access memory. For relatively simple CPU memory mapped I/O is used.
- There are three registers in ISA of this processor.
  - Accumulator (AC)
  - Register R
  - Zero flag (Z)

Simple Instruction Set

Instruction	Instruction Code	Operation
NOP	0000 0000	No operation
LDAC	0000 0001 Γ	$AC = M[\Gamma]$
STAC	0000 0010 Γ	$M[\Gamma] = AC$
MVAC	0000 0011	$R = AC$
MOVR	0000 0100	$AC = R$
JUMP	0000 0101 Γ	GOTO Γ
JMPZ	0000 0110 Γ	IF $(Z=1)$ THEN GOTO Γ
JPNZ	0000 0111 Γ	IF $(Z=0)$ THEN GOTO Γ
ADD	0000 1000	$AC = AC + R$, If $(AC + R = 0)$ Then $Z = 1$ Else $Z = 0$
SUB	0000 1001	$AC = AC - R$, If $(AC - R = 0)$ Then $Z = 1$ Else $Z = 0$
INAC	0000 1010	$AC = AC + 1$, If $(AC + 1 = 0)$ Then $Z = 1$ Else $Z = 0$
CLAC	0000 1011	$AC = 0, Z = 1$
AND	0000 1100	$AC = AC \wedge R$, If $(AC \wedge R = 0)$ Then $Z = 1$ Else $Z = 0$
OR	0000 1101	$AC = AC \vee R$, If $(AC \vee R = 0)$ Then $Z = 1$ Else $Z = 0$
XOR	0000 1110	$AC = AC \oplus R$, If $(AC \oplus R = 0)$ Then $Z = 1$ Else $Z = 0$
NOT	0000 1111	$AC = AC'$, If $(AC' = 0)$ Then $Z = 1$ Else $Z = 0$

Simple Instruction Set

- The final component is the instruction set architecture for this relatively simple CPU is shown in the table above.
- The LDAC, STAC, JUMP, JMPZ AND JPNZ instructions all require a 16-bit memory address represented by the symbol Γ.
- Since each byte of memory is 8-bit wide these instructions requires 3 bytes in memory. The first byte contains the opcode for the instruction and the remaining 2 bytes for the address.

Relatively Simple Computer

- In this relatively simple computer Figure 1:11 shown below, we put all the hard ware components of the computer together in one system. This computer will have 8K ROM starting at address 0 fallowed by 8K RAM. It also has a memory mapped bidirectional I/O port at address 8000H.
- First let us look at the CPU since it uses 16 bit address labeled A15 through A0. System bus via pins through D7 to D0. The CPU also has the two control lines READ and WRITE.
- Since it uses the memory mapped I/O it does not need a control signal such as .
- The relatively simple computer is shown in the figure below. It only contains the CPU details. Each part will be developed in the design.

Relatively Simple Computer

Relatively Simple CPU

A_{15}–A_0 16

D_7–D_0 8

READ

WRITE

Memory Subsystem

I/O Subsystem

Figure 1:11: A relatively simple computer: CPU details only

Relatively Simple Computer

- To access a memory chip the processor must supply an address used by the chip. An 8K memory chip has 213 internal memory locations it has 13bit address input to select one of these locations.

- The address input of each chip receives CPU address bits A12 to A0 from system address bus. The remaining three bits A15, A14, and A13 will be used to select one of the memory chips.

MODULE-II

ORGANIZATION OF A COMPUTER

Course Outcomes

CO 1	Describe Register transfer languages, arithmetic micro operations, logic micro operations, shift micro operations address sequencing, micro program example, and design of control unit.
CO 2	Classify the functionalities of various micro operations such as arithmetic, logic and shift micro operations.
CO 3	Describe the Control unit and Control memory operations.
CO 4	Knowledge about address sequencing in Control memory
CO 5	Explore the micro program example and design of control unit

Contents

Register transfer:

- Register transfer language
- Register transfer
- Bus and memory transfers
- Arithmetic micro operations
- Logic micro operations
- Shift micro operations

Control unit:

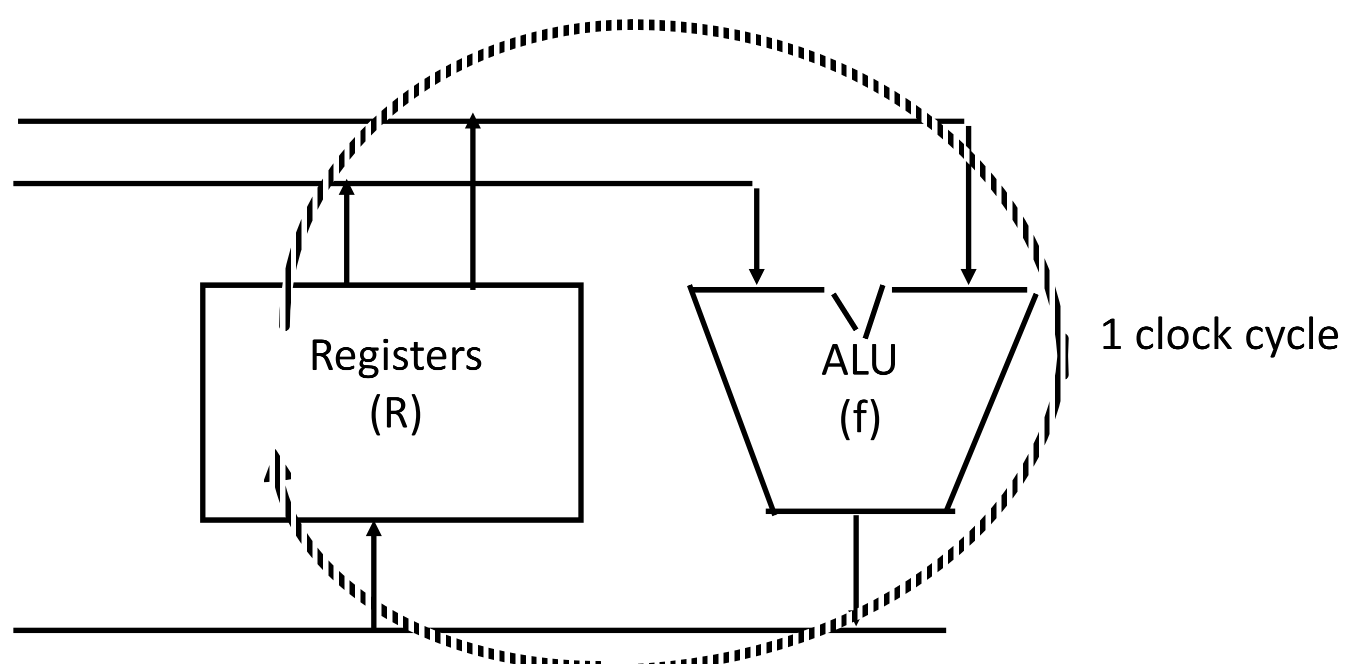
- Control memory
- Address sequencing
- Micro program example
- and Design of control unit.

Register Transfer Language

- A digital system is a Interconnection of digital hardware modules that accomplish a specific information processing task.
- Digital system uses a modular approach.
- The modules are constructed from such as digital components as registers ,decoders , arithmetic elements and control logic.
- The operations are performed on the data in the registers.
- The operations executed on the data in registers are called micro operations.
- The functions built into registers are examples of micro operations
  - Shift
  - Load
  - Clear
  - Increment …….

Register Transfer Language

- The Micro operation is an elementary operation performed (during one clock pulse)on the information stored in one or more registers.



- $R \leftarrow f(R, R)$
- f: shift, load, clear, increment, add, subtract, complement, and, or, xor, …

Register Transfer Language

- The internal organization of a computer is Defined as:
  - Set of registers and their functions
  - Micro operations set
    - Set of allowable microoperations provided by the organization of the computer
  - Control signals that initiate the sequence of micro operations (to perform the functions)

- The symbolic notation used to describe the micro operation transfers among registers is called a register transfer language.

Register Transfer

- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC - program counter
  - IR - instruction register
- Information Transfer from one register to another register is designated in symbolic form by using a replacement operator.

  $R2 \leftarrow R1$

- Registers and their contents can be viewed and represented in *various ways:*
  - A register can be viewed as a single entity:

MAR

Register Transfer

- Registers may also be represented showing the bits of data they contain

7	6	5	4	3	2	1	0

Showing individual bits

- Numbering of a registers

Numbering of bits

Register Transfer

- Portion of register

15 ... 8	7 ... 0
PC(H)	PC(L)

Subfields

Control Function

- The actions are performed only when certain conditions are true.
- This is similar to an “if” statement in a programming language.
- In digital systems, this is often done via a control signal, called a control function . control function is a Boolean variable.
  - If the signal is 1, the action takes place
- This is represented as:

  $P: R2 \leftarrow R1$

Register Transfer

- Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1) then (R2 ← R1).
- Implementation of controlled transfer

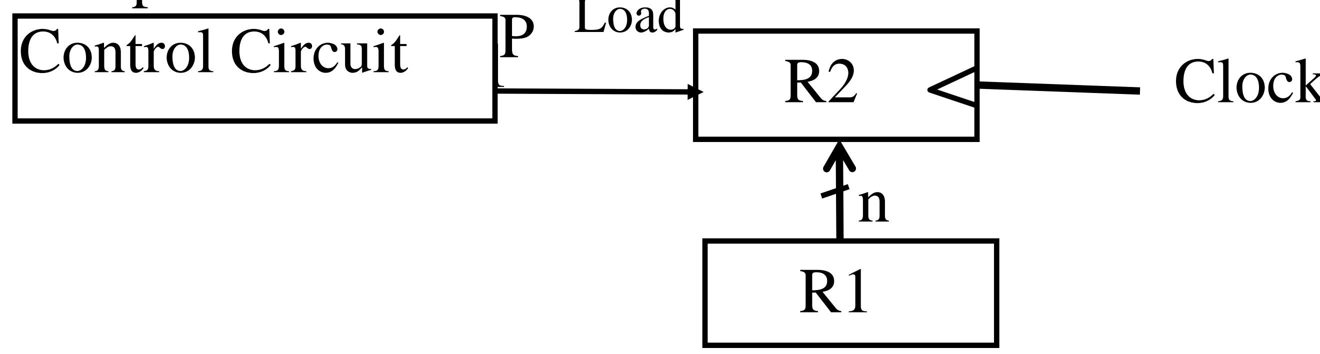


Fig: Block Diagram

Register Transfer

- The same clock controls the circuits that generate the control function and the destination register.

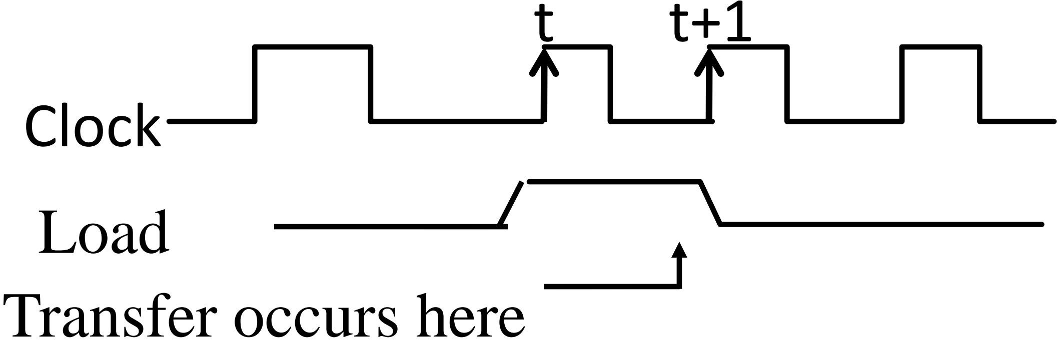


Fig :Timing Diagram

Register Transfer

- If two or more operations are to occur simultaneously, they are separated with commas

P: R3 ← R5,MAR ← IR

- Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR
- Basic Symbols Used for Register Transfer is

\- Letters and Numerals to denote a registers. Ex: MAR,IR,R2 .

\- Parentheses ( ) to denote a part of a register . Ex: R2(0-7),R2(L).

\- Arrow to denote transfer of Information. Ex: R2 R1

\- Comma , Separates two micro operations . Ex: R2← R1,R3←R4

Bus Transfer

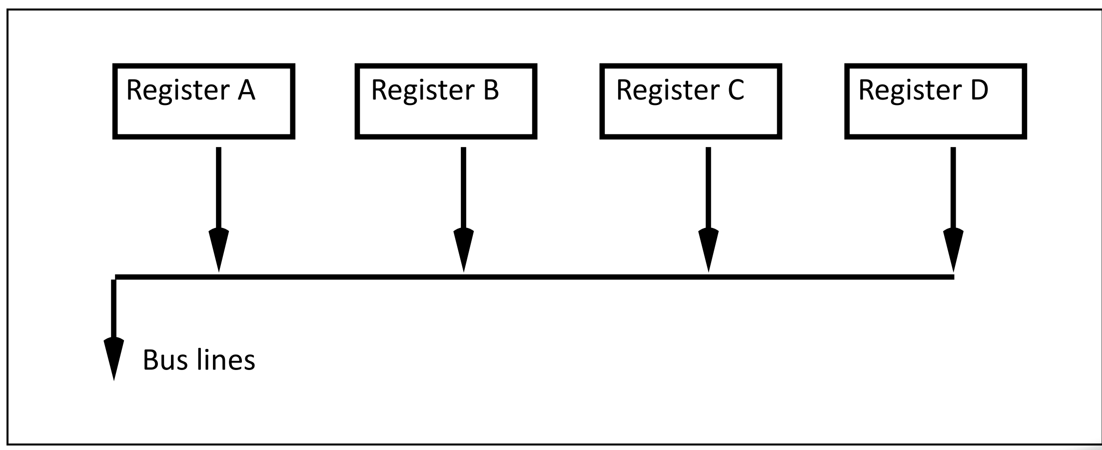
- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers

- To completely connect n registers → n(n-1) lines
- $O(n^2)$ cost
  - This is not a realistic approach to use in a large digital system

- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

Bus Transfer

- Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.
- One way to construct a common bus system is by using multiplexers.
- The multiplexer select the source register whose binary information is then placed into the bus.
- To transfer data from R1←C. i.e BUS←C , R1←BUS



Bus Transfer

S_1
S_0

4 × 1 MUX 3
4 × 1 MUX 2
4 × 1 MUX 1
4 × 1 MUX 0

4- line common bus

D_2 C_2 B_2 A_2
D_1 C_1 B_1 A_1
D_0 C_0 B_0 A_0

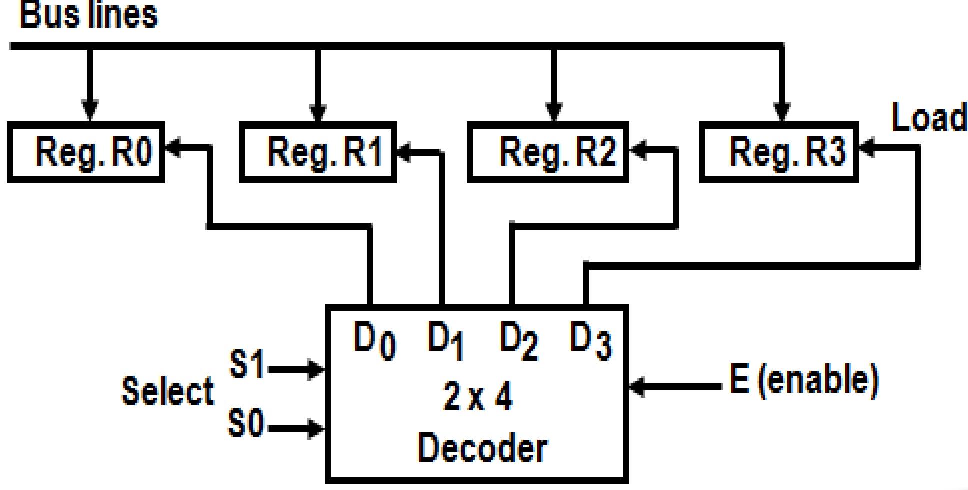
Register D
Register C
Register B
Register A

Bus Transfer

S1	S2	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Function Table For Bus

Fig : Transfer From Bus to Destination Register



Three State Bus Buffers

- A bus system can be implemented with three state gates instead of multiplexers.
- A three state gate is a digital circuit that exhibits three states.
- Two sates are normal signal states 1 and 0.The third state is a high impedance state.
- High impe ns the out put is ope

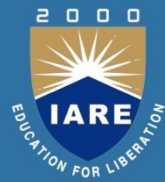
Normal input A

Control input C

Output Y=A if C=1
High-impedence if C=0

Fig: Graphic Symbol for Three State buffer

Three State Bus Buffers



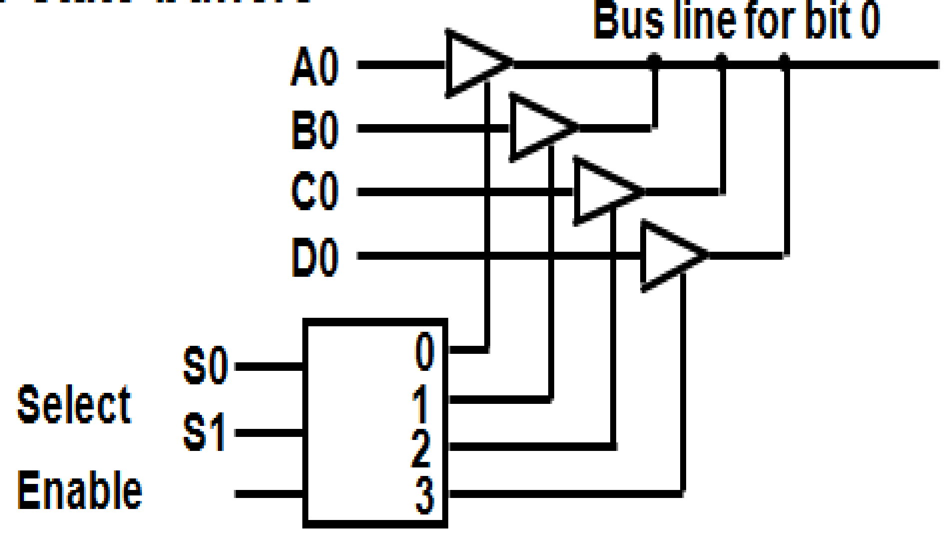


Fig : Bus Line with three state –Buffer

- To Construct a Common bus for four registers of n bits each using three state-bus buffer, we need n circuits with four buffers.

Memory Transfer

- The transfer of information from memory word to the outside environment is called Memory read operation.
- The transfer of new information to be stored into the memory is called memory write operation.
- Memory word is symbolically represented by using letter M.

Read : $DR \leftarrow M[AR]$

Write : $M[AR] \leftarrow R1$

SUMMARY OF R. TRANSFER MICROOPERATIONS

$A \leftarrow B$	Transfer content of reg. B into reg. A
$AR \leftarrow DR(AD)$	Transfer content of AD portion of reg. DR into reg. AR
$A \leftarrow$ constant	Transfer a binary constant into reg. A
$ABUS \leftarrow R1,$	Transfer content of R1 into bus A and, at the same time,
$R2 \leftarrow ABUS$	transfer content of bus A into R2
AR	Address register
DR	Data register
M[R]	Memory word specified by reg. R
M	Equivalent to M[AR]
$DR \leftarrow M$	Memory *read* operation: transfers content of memory word specified by AR into DR
$M \leftarrow DR$	Memory *write* operation: transfers content of DR into memory word specified by AR

Micro Operations

- Micro operation is an elementary operation performed with the data stored in registers.
- Computer system micro operations are of four types:
  1. Register transfer micro operations
  2. Arithmetic micro operations
  3. Logic micro operations
  4. Shift micro operations

- In register transfer micro operations the contents of the register can not be altered when transfer the data from the source to destination.
- The remaining three micro operations alter the data when transfer the data from one place to another.

Micro Operations

Arithmetic Micro Operations

- The basic arithmetic micro operations are addition , subtraction , increment , decrement and shift operations.
- R1←R2+R3 this micro operation specifies an add operation .
- It states that the contents of register R2 and R3 are added and result stored into register R1.
- R1←R2+R3+1(R2-R3) this micro operation specifies a subtract operation .
- The subtract operation is implemented through complementation and addition.

Micro Operations

Arithmetic Micro Operations

Microoperation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement the contents of R2
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	subtraction
$R1 \leftarrow R1 + 1$	Increment
$R1 \leftarrow R1 - 1$	Decrement

Binary Adder

- Basic hardware required to implement add operation is registers and digital component that perform add operation.
- The digital circuit that forms the arithmetic sum of two bits and previous carry is called a full adder.
- The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder.
- The binary adder is constructed with full adder circuits connected in cascade , with the out put carry from one full adder connected to the input carry of the next full adder.
- An n bit binary adder requires n full adders.
- The output carry from each full adder is connected to the input carry of the next-higher –order full adder.

Micro Operations

B3 A3 B2 A2 B1 A1 B0 A0

C3 C2 C1 C0

FA FA FA FA

C4 S3 S2 S1 S0

Fig: 4-bit binary adder

Binary Adder- Sub tractor

- The Subtraction of A-B can be done by taking the 2’s Complement of B and adding it to A.
- The addition and subtraction can be implemented by one common

B3 A3 B2 A2 B1 A1 B0 A0
M
FA C3 FA C2 FA C1 FA C0
C4 S3 S2 S1 S0

Fig: 4-bit Adder- subtractor

Binary Adder- Sub tractor

- Here the mode input M controls the operation.
- When M=0 the circuit works like an adder and M=1 the circuit works like a subtractor .
- Each exclusive-OR gate receives input M and one of the inputs of B.
- When M=0 the operation specifies $B \oplus 0=B$.
- The full-adder receives the value of B ,the input carry is 0 ,and the circuit performs A plus B.
- When M=1 the operation specifies $B \oplus 1=B'$ and $C_0=1$.
- The B inputs are all complemented and 1 is added through the input carry .
- Then the circuit performs operation A +2's Complement of B.
- Unsigned Numbers it gives A-B if A>=B or the 2's complement of (B-A)if A<B.
- Signed Numbers the result is A-B provided that there is no overflow.

Binary Incrementer

- The increment micro operation adds one to a number in a register.
- This is implemented by using a binary counter.
- The binary counter is implemented by using half adders connected in cascade.

A3 A2 A1 A0 1

x y HA C S | x y HA C S | x y HA C S | x y HA C S

C4 S3 S2 S1 S0

Fig : 4- bit binary Incrementer

Binary Incrementer

- One of the inputs to the least significant half adder (HA)is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
- The output carry from one half adder is connected to one of the inputs of the next-higher-order half adder.
- The generated out is displayed in s0 through s3.
- The above circuit can be implemented to an n-bit binary incermenter by including n-half adders.

Arithmetic Circuit

- The arithmetic operation can be implemented by using single composite arithmetic circuit.
- The basic component of an arithmetic circuit is parallel adder. By controlling the input of the parallel adder we obtain the different types of arithmetic operations.
- In a 4-bit Arithmetic circuit it contains 4 full adders and four multiplexers to choose different operations.
- There are two 4-bit inputs A and B and a 4-bit output D.
- The four Inputs From A directly connected to Adder and the B input is given to input of Multiplexers.
- The multiplexers data also receives the complement of B .
- The remaining two data inputs are connected to the Logic-0 and 1.
- The four multiplexers are controlled by two selection inputs , S1 and S0.

Arithmetic Circuit

Fig: 4-bit Arithmetic circuit

Arithmetic Circuit

- The input carry Cin goes to the carry input of the FA in the least significant position .
- The other carries a connected from one stage to the next.
- The output of the Binary Adder is calculated by using Arithmetic sum

  D= A+Y+Cin

  here A is 4-bit Binary number at X inputs.

  Y is 4-bit binary number at Y inputs of the binary adder.

  Cin is the input carry.
- In the above equation by controlling the value of Y with two control inputs S1 and S0 and making the Cin 1 or 0 the above circuit performs all the arithmetic operations.

Arithmetic Circuit

S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	$\overline{B}$	D = A + $\overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	D = A + $\overline{B}$+ 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

Logic Micro operations

- Logic micro operations specify the operations for strings of bits stored in registers.
- Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data.
- The logic microoperation exclusive-OR with the contents of two registers R1 and R2 is symbolized by the statement:

$$P : R1 \leftarrow R1 \oplus R2$$

Ex: R1=1010 R2=1100

if P=1 then

1010 =R1

1100 =R2

0110 =R1 after P=1

Logic Micro operations

Name	Graphic symbol	Algebraic function	Truth table
AND	(AND gate: inputs A, B; output x)	$x = A \cdot B$ or $x = AB$	A B | x: 0 0 | 0; 0 1 | 0; 1 0 | 0; 1 1 | 1
OR	(OR gate: inputs A, B; output x)	$x = A + B$	A B | x: 0 0 | 0; 0 1 | 1; 1 0 | 1; 1 1 | 1
Inverter	(Inverter: input A; output x)	$x = A'$	A | x: 0 | 1; 1 | 0
Buffer	(Buffer: input A; output x)	$x = A$	A | x: 0 | 0; 1 | 1

Logic Micro operations

Name	Algebraic function	A	B	x
NAND	$x = (AB)'$	0	0	1
		0	1	1
		1	0	1
		1	1	0
NOR	$x = (A + B)'$	0	0	1
		0	1	0
		1	0	0
		1	1	0
Exclusive-OR (XOR)	$x = A \oplus B$ or $x = A'B + AB'$	0	0	0
		0	1	1
		1	0	1
		1	1	0
Exclusive-NOR or equivalence	$x = (A \oplus B)'$ or $x = A'B' + AB$	0	0	1
		0	1	0
		1	0	0
		1	1	1

Logic Micro operations

- Special Symbols used for logic microoperations OR , AND and Complement.

  OR = V , AND = ^ and Complement = ___

- The main aim of adopting two sets of symbols is to differentiate between logic microoperations and control (Boolean)functions.

- P+Q : R1←R2+R3 , R4 ← R5 V R6

  In the above statement + symbol performs OR operation between P and Q .It performs arithmetic addition between R2 and R3 .

Logic Micro operations

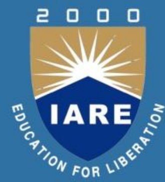
List Of Logic Microoperations

- There are 16 different microoperations that can be performed with two binary operations.
- Most of the systems implement four of these ^,V,__ and ⊕.

X	Y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table :Truth Table for 16 functions of Two Variables

Logic Micro operations



Boolean Function	Micro-Operations	Name
$F0 = 0$	$F \leftarrow 0$	Clear
$F1 = xy$	$F \leftarrow A \wedge B$	AND
$F2 = xy'$	$F \leftarrow A \wedge B'$	
$F3 = x$	$F \leftarrow A$	Transfer A
$F4 = x'y$	$F \leftarrow A' \wedge B$	
$F5 = y$	$F \leftarrow B$	Transfer B
$F6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F7 = x + y$	$F \leftarrow A \vee B$	OR
$F8 = (x + y)'$	$F \leftarrow (A \vee B)'$	NOR
$F9 = (x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
$F10 = y'$	$F \leftarrow B'$	Complement B
$F11 = x + y'$	$F \leftarrow A \vee B'$	
$F12 = x'$	$F \leftarrow A'$	Complement A
$F13 = x' + y$	$F \leftarrow A' \vee B$	
$F14 = (xy)'$	$F \leftarrow (A \wedge B)'$	NAND
$F15 = 1$	$F \leftarrow$ all 1's	Set to all 1's

Table : 16 Microoperations

Hardware Implementation

- To implement these microoperations it uses Logic gates.
- Most of the computers implement only four functions like AND , OR, NOT and XOR .

Figure 10 One stage of logic circuit.

S_1 S_0 4 × 1 MUX A_i B_i 0 1 2 3 E_i

(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

Logic Microoperations

Applications

- Logic microoperations can be used to manipulate individual bits or a portions of a word in a register
- Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A

  - Selective-set $A \leftarrow A + B$
  - Selective-complement $A \leftarrow A \oplus B$
  - Selective-clear $A \leftarrow A \bullet B'$
  - Mask (Delete) $A \leftarrow A \bullet B$
  - Clear $A \leftarrow A \oplus B$
  - Insert $A \leftarrow (A \bullet B) + C$
  - Compare $A \leftarrow A \oplus B$

Logic Micro operations

Selective-Set

- The Selective-Set Operation sets to 1 the bits in register A where there are corresponding 1's in Register B.
- It does Not affect bit positions that have 0 in B.

1 1 0 0	A(before)
1 0 1 0	B(logic operand)
1 1 1 0	A (after) $(A \leftarrow A + B)$

- The OR microoperation can be used to selective set bits of a Register.

Logic Micro operations

Selective-Complement

- The selective-complement operation complements bits in register A where there are corresponding 1's in B.
- It does not affect the bit positions that have 0's in B.

```
  1 0 1 0  A  (A before)
  1 1 0 0  B  (Logic Operand)
  -----------
  0 1 1 0  A  ( A after)     (A ← A ⊕ B)
```

- The exclusive microoperation can be used to selectively complement bits of register.

Logic Micro operations

Selective-Clear

- The Selective-clear operation clears to0 the bits in A only where there are corresponding 1's in B .

1 1 0 0	A	(A before)
1 0 1 0	B	(Logical Operand)
0 1 0 0	A	($A \leftarrow A$ ^B')

- The Boolean operation performed on the individual bits in AB'.

Logic Micro operations

Mask

- The Mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B.
- The mask operation is similar to an AND microoperation.

1 1 0 0	A	(Before)
1 0 1 0	B	(Logical Operand)
1 0 0 0	A	$(A \leftarrow A \cdot B)$

Logic Microoperations

Insert

- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions

Example

- Suppose you wanted to introduce 1010 into the low order four bits of A:
  1101 1000 1011 0001 A (Original)
  1101 1000 1011 1010 A (Desired)

```
 1101 1000 1011 0001        A (Original)
----------------------------------------
1111 1111 1111 0000         Mask
1101 1000 1011 0000         A (Intermediate)
----------------------------------------
0000 0000 0000 1010         Added bits
1101 1000 1011 1010         A (Desired)
```

Logic Microoperations

Clear

- The Clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal.
- This operation is achieved by exclusive-OR operation.

1 1 0 0 A	(Before)
1 1 0 0 B	(Logic Operand)
0 0 0 0 A	$(A \leftarrow A \oplus B)$

Shift Microoperations

- Shift micro operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations.
- The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.
- A logical shift is one that transfers 0 through the serial input. We will adopt the symbols SHL and SHR for logical shift-left and shift-right micro operations. For example:
-
- R1 ← SHL R1
- R2 ← SHR R2

Shift Microoperations

TABLE 7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Shift Microoperations

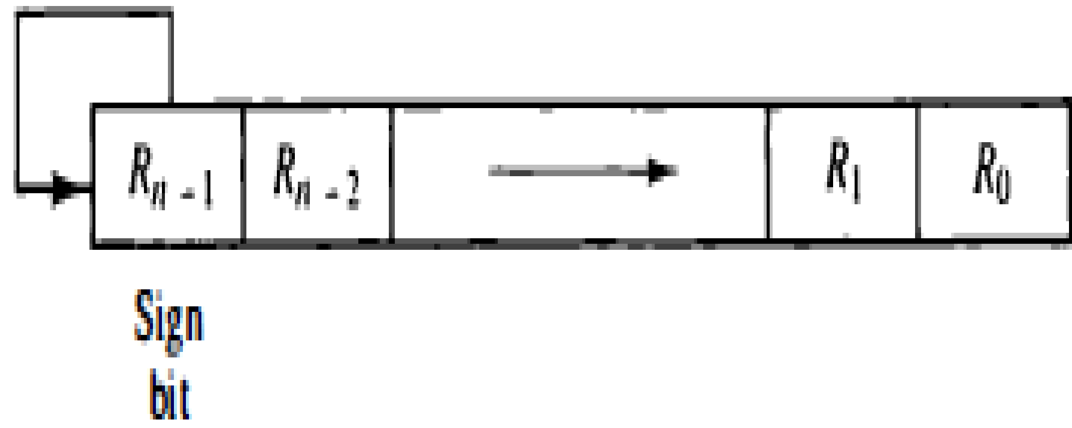
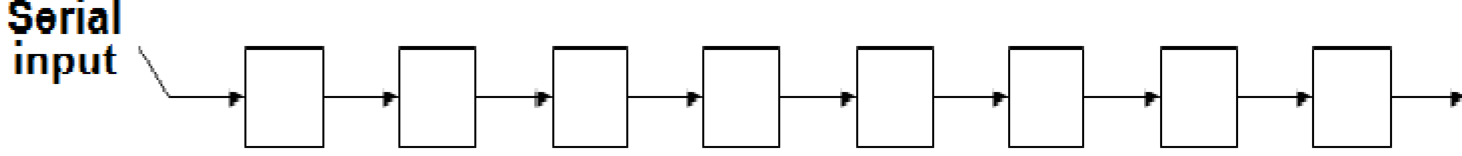


Figure 11 Arithmetic shift right.

Shift Microoperations

- The contents of a register can be shifted to right or left.

  - A right shift operation



  - A left shift operation

Serial input

Shift Microoperations

- The information transferred through the serial input determines the type of shift.
- There are three types of shifts
  - Logical shift
  - Circular shift
  - Arithmetic shift

Logical Shift Microoperations

- In a logical shift the serial input to the shift is a 0.

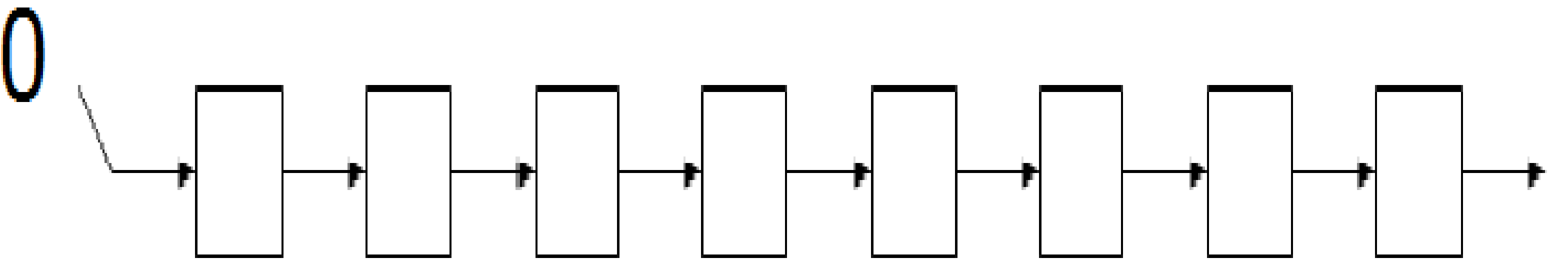


Fig : Right Logical Shift Operation

Shift Microoperations

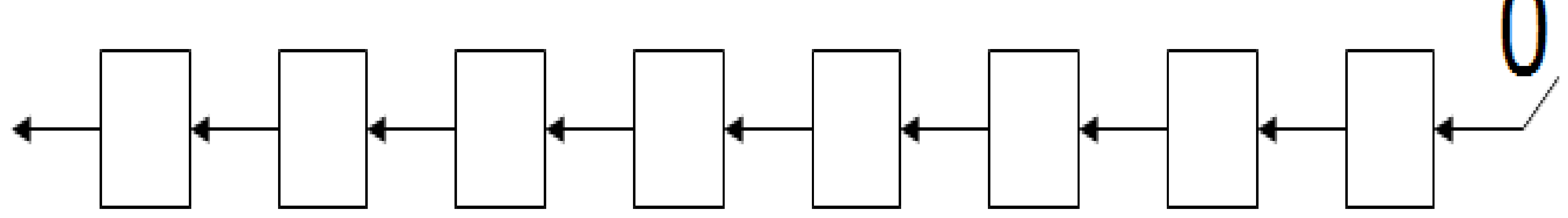


Fig: Left logical shift Microoperation

- Notations used to denote logical shift microoperations are:

  shl ----> for logical shift left

  shr ----> For logical shift right

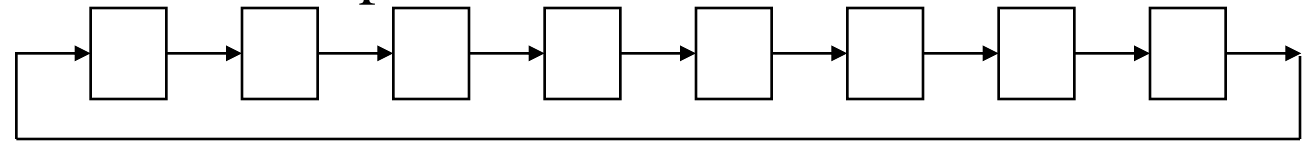
Examples :

R2 ← shl R2

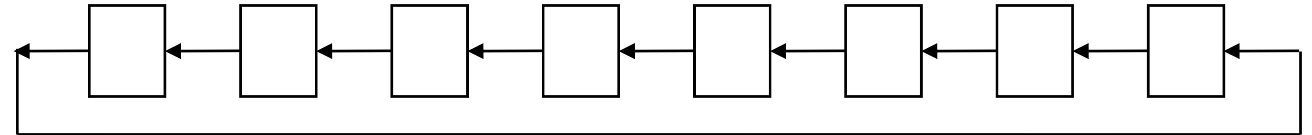
R3 ← shr R3

Shift Microoperations

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.
- A right circular shift operation:



- A left circular shift operation:

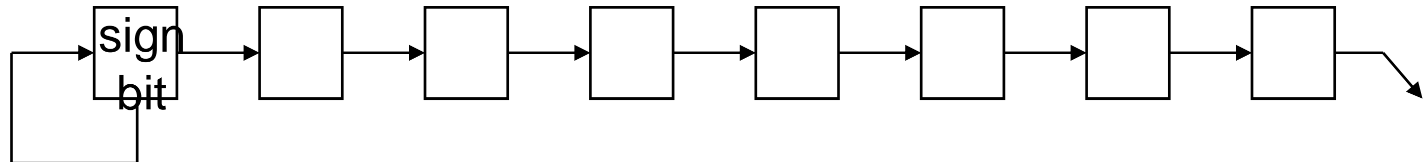


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← *cir* R2
    - R3 ← *cil* R3

Shift Microoperations

Arithmetic Shift

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift multiplies a signed number by two
- An arithmetic right shift divides a signed number by two
- The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division
- A right arithmetic shift operation:

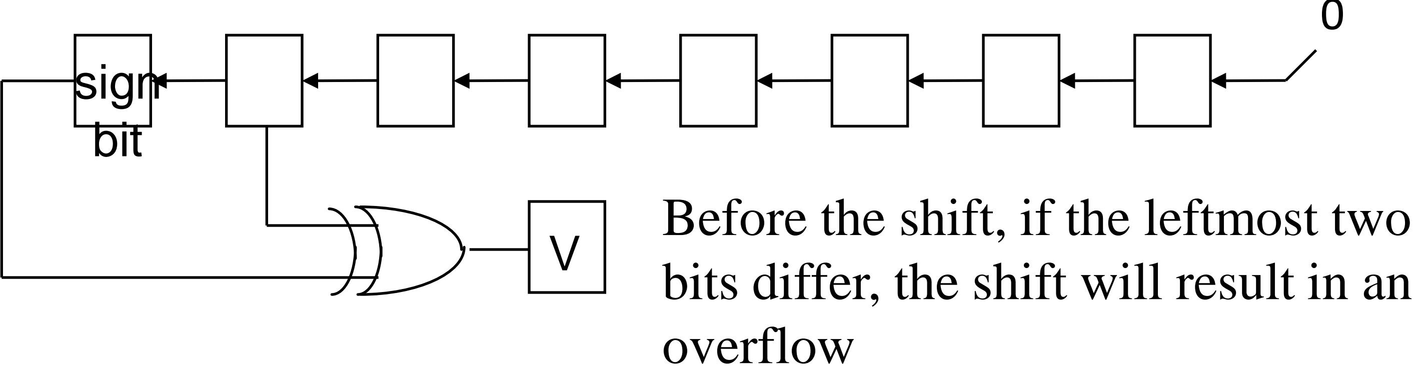


- A left arithmetic shift operation:

0
sign
bit

Shift Microoperations

- An left arithmetic shift operation must be checked for the overflow



- In a RTL, the following notation is used
  - *ashl* for an arithmetic shift left
  - *ashr* for an arithmetic shift right
  - Examples:
    - » R2 ← *ashr* R2
    - » R3 ← *ashl* R3

Hardware Implementation

4-bit Combinational Shifter

Hardware Implementation

Select bit S	output			
	H0	H1	H2	H3
0	IR	A0	A1	A2
1	A1	A2	A3	IL

Table : Function Table

- When S=1 the input data is shifted to left
- When s=0 the input data is shifted to right.

Arithmetic Logic Shift Unit

- All the three operations are implemented with a single circuit.

S3
S2
S1
S0
C_i
Arithmetic Circuit
D_i
C_{i+1}
Select
0
1
2
3
4 x 1 MUX
F_i
Logic Circuit
E_i
B_i
A_i
A_{i-1}
shr
A_{i+1}
shl

-

Arithmetic Logic Shift Unit

S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B’	Subtract with borrow
0	0	1	0	1	F = A + B’+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	F = A’	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F

Control Memory

- Major Components of a digital system are CPU , Memory and I/O Devices.
- The major digital components of CPU are CU ,ALU and Registers.
- The function of control unit is to initiate sequence of micro operations.
- There are two ways to implement control unit:

1. Hardwired Control unit
2. Micro programmed Control Unit

Control Memory

Hardwired Control unit:

- When the control signals are generated by hardware using conventional logic design techniques then it is called hardwired control unit.
- Hardwired control implemented with fixed instructions , fixed logic blocks of arrays , encoders , decoders etc.
- The characteristics of Hardwired control logic are high speed operation , expensive , relatively complex and no flexibility to add new instructions.

Micro programmed CU:

- The main principle of microprogramming is an elegant and systematic method for controlling the micro operation sequence in a digital computer.
- Micro programmed control unit contains variable number of instructions.
- Easy to implement and add new instructions.

Control Memory

- The Control variable at any given time can be represented by a string of 1's and 0's called a Control Word.
- **Microprogram**
  - Program stored in memory that generates all the control signals required to execute the instruction set correctly.
  - Consists of microinstructions .
- **Microinstruction**
  - It Specifies one or more micro operations for the system.
  - Contains a control word and a sequencing word .
  - Control Word - All the control information required for one clock cycle
  - **Sequencing Word** - Information needed to decide the next microinstruction address .
- **Control Memory**.

-Storage in the microprogrammed control unit to store the microprogram.

Control Memory

- **Writeable Control Memory(Writeable Control Storage:WCS)**
  - CS whose contents can be modified
    - Allows the microprogram can be changed
    - Instruction set can be changed or modified .
- **Dynamic Microprogramming**
    - Computer system whose control unit is implemented with a microprogram in WCS .
    - Microprogram can be changed by a systems programmer or a user .
- **Sequencer (Microprogram Sequencer)**

 - A Microprogram Control Unit that determines the Microinstruction Address to be executed in the next clock cycle .

Control Memory

- In-line Sequencing
- Branch
- Conditional Branch
- Subroutine
- Loop
- Instruction OP-code mapping

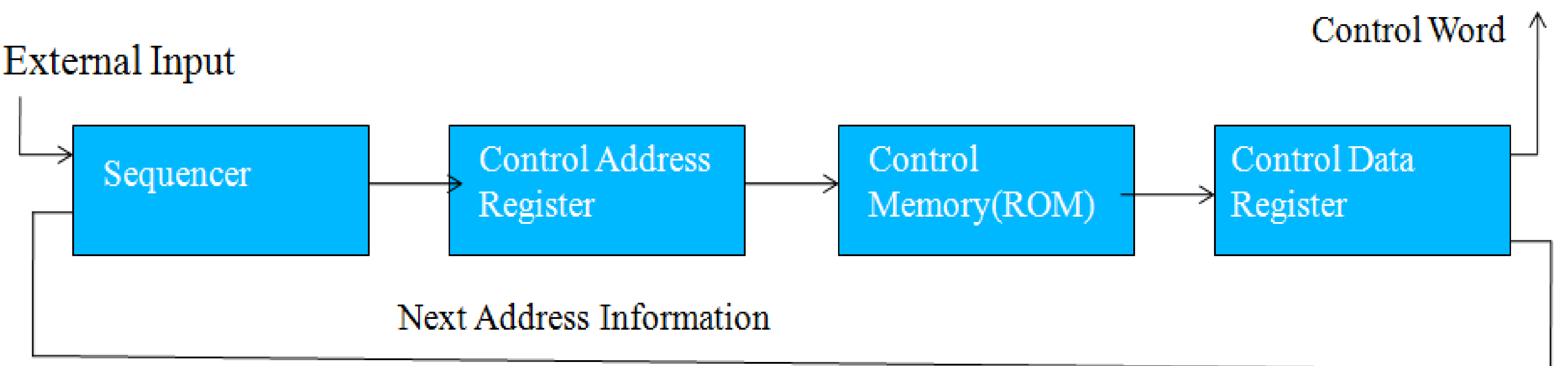


Fig : Microprogrammed Control Unit

Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a routine.

- Each computer instruction has its own micro program routine in control memory to generate the micro operations that execute the instruction.

- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.

Address Sequencing

- This address is usually the address of the first microinstruction that activates the instruction fetch routine.

- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.

- At the end of the fetch routine, the instruction is in the instruction register of the computer

Address Sequencing

The address sequencing capabilities required in a control memory are:

- 1. Incrementing of the control address register.
- 2. Unconditional branch or conditional branch, depending on status bit conditions.
- 3. A mapping process from the bits of the instruction to an address for control memory

Address Sequencing

Instruction code

Mapping logic

Status bits

Branch logic

MUX select

Multiplexers

Subroutine register (SBR)

Clock

Control address register (CAR)

Incrementer

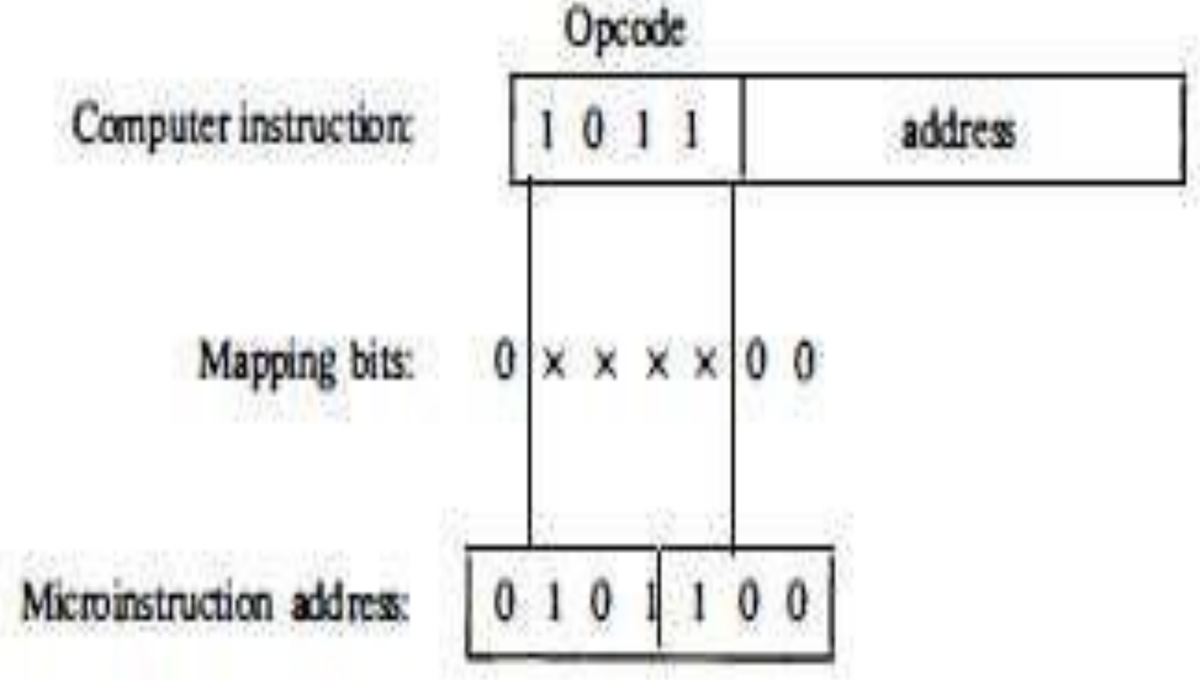
Control memory

Select a status bit

Microoperations

Branch address

Address Sequencing



Address Sequencing

- **Subroutines**
- Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the micro program.
- Frequently, many micro programs contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.

Address Sequencing

- This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.
- Micro programs that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return.

Address Sequencing

MUX
10 0
AR
Address
Memory
2048 x 16
10 0
PC
MUX
6 0
SBR
6 0
CAR
15 0
DR
Control memory
128 x 20
Arithmetic
logic and
shift unit
Control unit
15 0
AC

Micro program Example

- A micro program sequencer can be constructed with digital functions to suit a particular application.

- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.

- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- The binary values of the two selection variables determine the path in the multiplexer.

- For example, with S1, So = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

Micro program Example

External (MAP)
3 2 1 0
I_0 I_1 T Input logic
L
S_1 S_0 MUX 1
Load
SBR
Incrementer
1 I S Z
MUX 2 Select
Test
Clock
CAR
Control memory
Microops CD BR AD

Design of Control Unit

- The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function.

- The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching.

- The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2' microoperations.

Design of Control Unit

- Figure 2.18 shows the three decoders and some of the connections that must be made from their outputs.

- Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 x 8 decoder to provide eight outputs.

- As shown in Figure 2.18 outputs 5 and 6 of decoder f1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.

Design of Control Unit

TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer

BR Field		Input I_1	I_0	T	MUX 1 S_1	S_0	Load SBR L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

Design of Control Unit

- The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive, as shown in Figure 2.18. The other outputs of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

Design of Control Unit

F1

F2

F3

3 × 8 decoder

7 6 5 4 3 2 1 0

3 × 8 decoder

7 6 5 4 3 2 1 0

3 × 8 decoder

7 6 5 4 3 2 1 0

AND

ADD

DRTAC

Arithmetic logic shift unit

PCTAR

DRTAR

From *PC*

From *DR* (0 – 10)

Load

AC

0

1

Select

Multiplexers

Load

AR

Clock

Design of Control Unit

- A microprogram sequencer can be constructed with digital Functions to suit a particular application.
- However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units.
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.
- The block diagram of the microprogram sequencer is shown in Fig.

Design of Control Unit

- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.

- There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR .

Design of Control Unit

External (MAP)
3 2 1 0
I_0 I_1 T Input logic
L
S_1 S_0 MUX 1
SBR
Load
Incrementer
1 I S Z
MUX 2 Select
Test
Clock
CAR
Control memory
Microops CD BR AD

Design of Control Unit

- The input logic circuit in Fig. has three inputs, l0, l1, and T, and three outputs, S0, S1, and L.Variables So and S, select one of the source addresses for CAR .
- Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer.
- For example, with S1 So = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

MODULE-III

CPU AND COMPUTER ARITHMETIC

Course Outcomes

CO 1	Understand the Instruction cycles and data representation in CPU design .
CO 2	Classify the input-output and interrupt, addressing modes.
CO 3	Describe the data transfer and manipulation and program control in CPU design .
CO 4	Knowledge about Addition and subtraction and floating point arithmetic operations in Computer arithmetic.
CO 5	Understand the decimal arithmetic unit Computer arithmetic.

Contents

CPU design:

- **Instruction cycle**
- **Data representation**
- **Memory reference instructions**
- **Input-output and interrupt**
- **Addressing modes**
- **Data transfer and manipulation**
- **Program control.**

Computer arithmetic:

- **Addition and subtraction**
- **Floating point arithmetic operations**
- **Decimal arithmetic unit.**

Instruction Cycle

- Each program is a sequence of instructions.
- The basic computer system each instruction is subdivided into Four phases:

1. Fetch an Instruction from memory.

2. Decode the Instruction.

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

4. Execute the Instruction.

- The above process continues indefinitely unless a HALT instruction is encountered.

Instruction Cycle

Fetch and Decode

- Initially the program counter PC is loaded with the address of the first instruction.

$T0: AR \leftarrow PC \ (S_0S_1S_2=010, T0=1)$

$T1: IR \leftarrow M[AR], \ PC \leftarrow PC + 1 \ \ (S0S1S2=111, T1=1)$

$T2: D0, \ldots, D7 \leftarrow \text{Decode } IR(12\text{-}14), AR \leftarrow IR(0\text{-}11), I \leftarrow IR(15)$

Instruction Cycle

T1
T0
S_2
S_1
S_0
Bus
Memory unit
7
Address
Read
AR
1
LD
PC
2
INR
IR
5
LD
Clock
Common bus

Fig: Register Transfer for the fetch phase

Instruction Cycle

- In the above diagram shows the transfer of first two statements(T0 and T1).
- When timing signal T0=1then

1. Place the contents of PC onto the bus by making the bus selection inputs S2S1S0 equal to 010.
2. Transfer the contents of the bus to AR by enabling the LD input of AR.

- When timing signal T1=1 then

1. Enable the read input of memory.
2. Place the contents of Memory onto the bus by making S2S1S0=111.
3. Transfer the contents of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

Instruction Cycle

Determine the Type of Instruction

- After executing the timing signal T1 the control unit determines the type of instruction that is read from memory.
- If D7=1 and the instruction must be a register-reference or input-out type.
- If D7=0 the operation code must be one of the other seven values 000 through 110 specifying a memory –reference Instruction.
- The symbolic representation is :

D'7IT3 :	AR ←M[AR]
D'7I'T3 :	Nothing
D7I'T3 :	Execute a register-reference instr.
D7IT3 :	Execute an input-output instr.

Instruction Cycle

Start
SC ← 0

T0
AR ← PC

T1
IR ← M[AR], PC ← PC + 1

T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)

D7

(Register or I/O) = 1

= 0 (Memory-reference)

I

(I/O) = 1

= 0 (register)

I

(indirect) = 1

= 0 (direct)

T3
Execute
input-output
instruction
SC ← 0

T3
Execute
register-reference
instruction
SC ← 0

T3
AR ← M[AR]

T3
Nothing

T4
Execute
memory-reference
instruction
SC ← 0

Fig : Flowchart for Instruction Cycle

Register Reference Instructions

- If D7=1 and I=0 then the instruction is recognized as register reference instruction.
- Register reference instructions use bits 0 to 11 of the instruction code to s
  11).

	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_{9}:	$AC \leftarrow AC'$
CME	rB_{8}:	$E \leftarrow E'$
CIR	rB_{7}:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$
CIL	rB_{6}:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$
INC	rB_{5}:	$AC \leftarrow AC + 1$
SPA	rB_{4}:	if $(AC(15) = 0)$ then $(PC \leftarrow PC+1)$
SNA	rB_{3}:	if $(AC(15) = 1)$ then $(PC \leftarrow PC+1)$
SZA	rB_{2}:	if $(AC = 0)$ then $(PC \leftarrow PC+1)$
SZE	rB_{1}:	if $(E = 0)$ then $(PC \leftarrow PC+1)$
HLT	rB_{0}:	$S \leftarrow 0$ (S is a start-stop flip-flop)

Memory-Reference Instructions

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

Table : Memory-Reference Instructions

- The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1.
- The execution of MR instruction starts with T_4

Memory-Reference Instructions

- AND to AC

D_0T_4: $DR \leftarrow M[AR]$ Read operand

D_0T_5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$ AND with AC

- ADD to AC

D_1T_4: $DR \leftarrow M[AR]$ Read operand

D_1T_5: $AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$ Add to AC and store carry in E

- LDA: Load to AC

D_2T_4 : $DR \leftarrow M[AR]$

D_2T_5 : $AC \leftarrow DR, SC \leftarrow 0$

- STA: Store AC

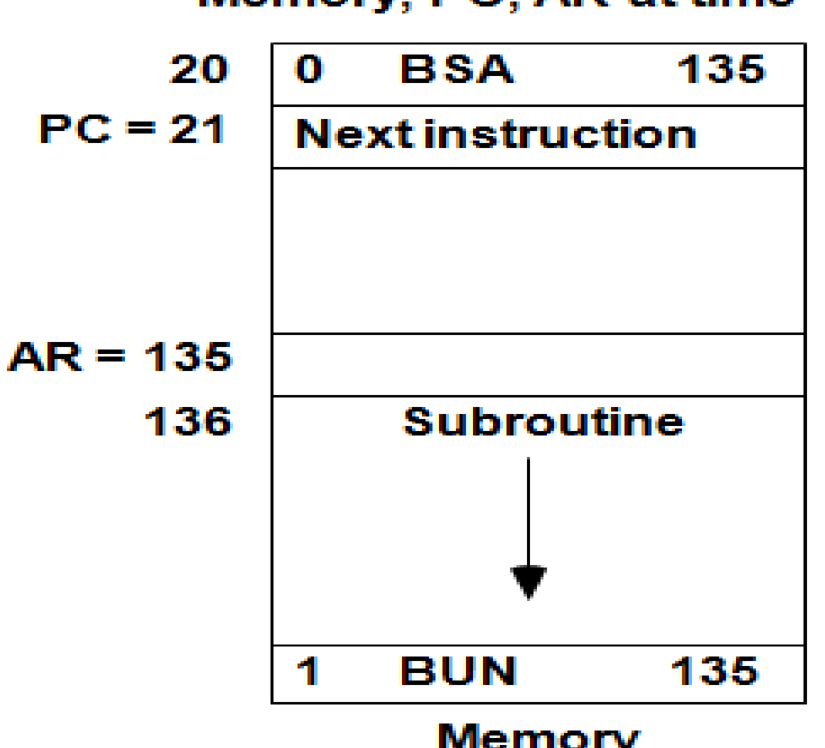
D_3T_4 : $M[AR] \leftarrow AC, SC \leftarrow 0$

Memory-Reference Instructions

- BUN: Branch Unconditionally

  D_4T_4: PC $\leftarrow$ AR, SC $\leftarrow$ 0

- BSA: Branch and Save Return Address



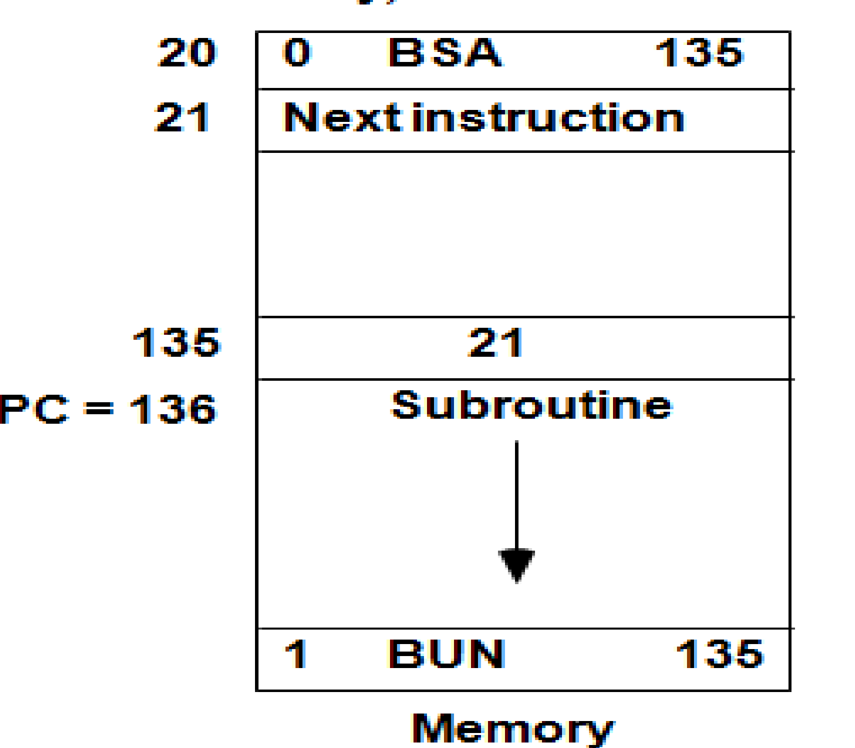


Fig: BSA Instruction Execution

Memory-Reference Instructions

- BSA:

  D_5T_4: $M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$

  D_5T_5: $PC \leftarrow AR,\ SC \leftarrow 0$

- ISZ: Increment and Skip-if-Zero

  $D_6T_4: DR \leftarrow M[AR]$

  $D_6T_5: DR \leftarrow DR + 1$

  $D_6T_4: M[AR] \leftarrow DR,\ \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1),\ SC \leftarrow 0$

Memory-Reference Instructions

Memory-reference instruction

AND
- D_0T_4: DR ← M[AR]
- D_0T_5: AC ← AC ∧ DR, SC ← 0

ADD
- D_1T_4: DR ← M[AR]
- D_1T_5: AC ← AC + DR, E ← Cout, SC ← 0

LDA
- D_2T_4: DR ← M[AR]
- D_2T_5: AC ← DR, SC ← 0

STA
- D_3T_4: M[AR] ← AC, SC ← 0

BUN
- D_4T_4: PC ← AR, SC ← 0

BSA
- D_5T_4: M[AR] ← PC, AR ← AR + 1
- D_5T_5: PC ← AR, SC ← 0

ISZ
- D_6T_4: DR ← M[AR]
- D_6T_5: DR ← DR + 1
- D_6T_6: M[AR] ← DR, If (DR = 0) then (PC ← PC + 1), SC ← 0

Fig: Flowchart for Memory Reference Instructions

Data Representation

Decimal Number System

- The decimal number system in every day use employs the radix 10 system.
- The 10 symbols are 0,1,2,3,4,5,6,7,8 and 9.
- The string of digits 834.5 is interpreted as:

  $8X10^2 + 3X10^1 + 4X10^0 + 5X10^{-1} = 834.5$

Binary Number System

- Binary number system uses the radix 2.
- The two digit symbols used are 0 and 1.
- The string of symbols 1001 is interpreted as:

  $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8+0+0+1=9$

Octal Number System

- Octal Number System uses radix 8.
- The Symbols used to represent the octal number system is 0,1,2,3,4,5,6 and 7.
- The octal number is converted into decimal number system by forming the sum of the weighted digits.

Ex:

$(736.4)_8 = ?$

$= 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$

$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$

Hexadecimal Number System

- The hexadecimal number system uses radix 16.
- The symbols used to represent the hexadecimal number system is 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F.
- The hexadecimal number is converted into decimal number system by forming the sum of the weighted digits.

Ex:

$(F3)_{16} = ?$

$= F \times 16^1 + 3 \times 16^0$

$= 15 \times 16 + 3 = (243)_{10}$

Decimal to Other Number Systems

- Conversion from decimal to its equivalent representation in the radix r system is carried our by separating the number into its integer part and fraction part and converting each part separately.
- The conversion of a decimal integer into a base r representation is done by successive divisions by r and accumulation of the reminders.
- The conversion of a decimal fraction to radix r representation is accomplished by successive multiplication by r and accumulation of the integer digits obtained.

Decimal to Binary Conversion:

Ex: $(41.6875)_{10} = (101001.1011)_2$

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 0.6875

0.6875
x 2
1.3750
x 2
0.7500
x 2
1.5000
x 2
1.0000

$(0.6875)_{10} = (0.1011)_2$

Binary to Octal and Hexadecimal Conversion

- Each octal digit corresponds to three binary numbers i.e $8=2^3$.
- Each hexadecimal digit corresponds to four binary numbers i.e $16=2^4$.

Binary, octal, and hexadecimal conversion

1 2 7 5 4 3 Binary

1 0 1 0 1 1 1 1 0 1 1 0 0 0 1 1 Octal

A F 6 3 Hexa

BCD:

– BCD is used to represent the decimal numbers system to binary number system

Complement Of Numbers

Two types of complements for base R number system:

1) (R-1)'s complement

2) R's complement

1) The (R-1)'s Complement

- Number N in base r having n digits (r-1)'s complement is defined as $(r^n - 1) - N$.
- Subtract each digit of a number from (R-1)

Example

- 9's complement of 835_{10} is 164_{10}
- 1's complement of 1010_2 is 0101_2(bit by bit complement operation)

Complement Of Numbers

2) The R's Complement

- The r’s complement of an n-digit number N in base r is defined as r^n-N for N is not 0.
- Add 1 to the low-order digit of its (R-1)'s complement

Example

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 1010_2 is $0101_2 + 1 = 0110_2$

Input-Output and Interrupt

- A Terminal with a keyboard and a Printer.

Input-output terminal

Serial communication interface

Computer registers and flip-flops

Printer

Receiver interface

OUTR

FGO

AC

Keyboard

Transmitter interface

INPR

FGI

Serial Communications Path

Parallel Communications Path

Fig: Input-Output Configuration

Input-Output and Interrupt

- The terminal sends and receives serial information.
- The serial info. from the keyboard is shifted into INPR .
- The serial info. for the printer is stored in the OUTR.
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to *synchronize* the timing difference between I/O device and the computer.

Input-Output and Interrupt

$D_7IT_3 = p$

$IR(i) = B_i$, $i = 6, \ldots, 11$

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$, $FGI \leftarrow 0$	Input char. to AC
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7)$, $FGO \leftarrow 0$	Output char. from AC
SKI	pB_9:	if$(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	if$(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Fig: Input-Output Instruction

Program Interrupt

- The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.
- If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

Program Interrupt

R = Interrupt f/f

Instruction cycle
=0
R
=1
Interrupt cycle

Fetch and decode instructions

Store return address in location 0
$M[0] \leftarrow PC$

Execute instructions

IEN
=0
=1

Branch to location 1
$PC \leftarrow 1$

FGI
=1
=0

FGO
=1
=0

$IEN \leftarrow 0$
$R \leftarrow 0$

$R \leftarrow 1$

Fig: Flow chart for Interrupt Cycle

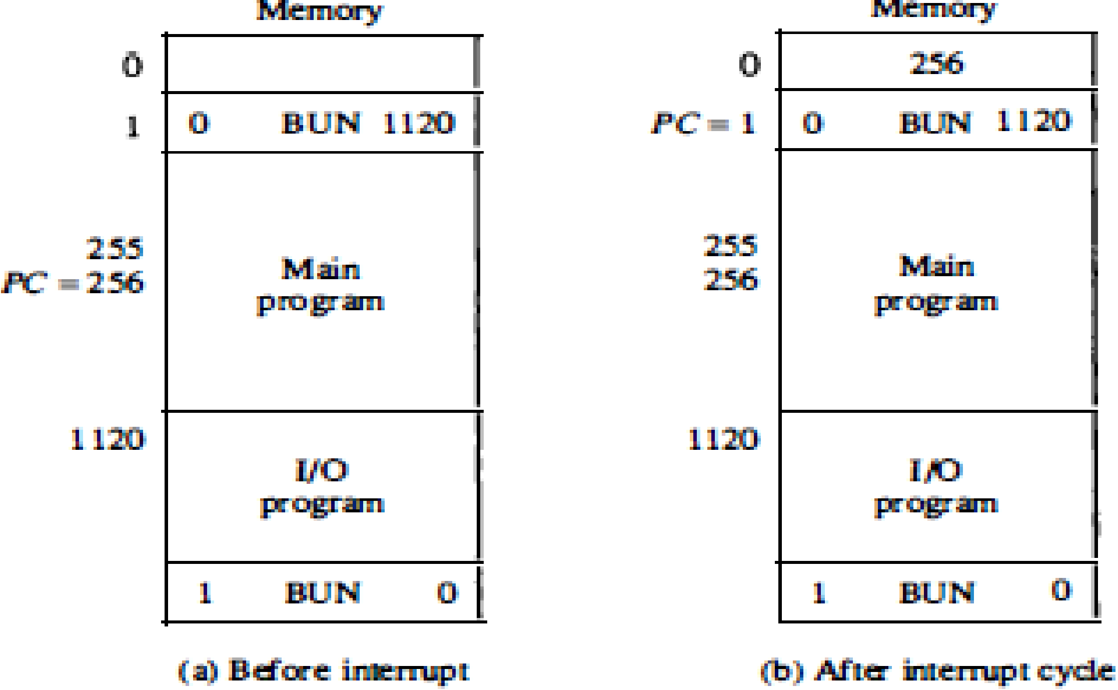
Program Interrupt

- An example that shows what happens during the interrupt cycle is shown in Fig. Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig(a).
- When control reaches timing signal T_0 and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120.

Program Interrupt

This program checks the flags, determines which flag is set, and then transfers the required input or output information. One this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. (b).

- **Figure:** Demonstration of the interrupt cycle



Addressing Modes

- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction .
- Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced).

Need for different addressing modes

1. To give programming flexibility to the user i.e pointers to memory, Counters for loop control,indexing of data,program relocation.
2. To use the bits in the address field of the instruction efficiently i.e reduce the number of bits.

Addressing Modes

1.Implied Mode

- Address of the operands are specified implicitly in the definition of the instruction.
- No need to specify address in the instruction.
- EA = AC, or EA = Stack[SP].
- Examples CLA, CME, INP

2. Immediate Mode

- Instead of specifying the address of the operand,operand itself is specified.
- No need to specify address in the instruction
- Operand itself needs to be specified
- Fast to acquire an operand

Addressing Modes

3.Register Mode

- Address specified in the instruction is the register address
- Designated operand need to be in a register
- Shorter address than the memory address
- Saving address field in the instruction
- Faster to acquire an operand than the memory addressing
- EA = IR(R) (IR(R): Register field of IR)

4.Register Indirect Mode

- Instruction specifies a register which contains the memory address of the operand .
- Saving instruction bits since register address is shorter than the memory address.
- Slower to acquire an operand than both the register addressing or memory addressing.
- EA = [IR(R)] ([x]: Content of x).

Addressing Modes

4.Autoincrement or Auto decrement Mode

- When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically .

5.Relative Addressing Modes

- The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand.
- Address field of the instruction is short.
- Large physical memory can be accessed with a small number of address bits.
- EA = f(IR(address), R)

Addressing Modes

- 3 Different Relative Addressing Modes depending on R;
  1. PC Relative Addressing Mode (R = PC)

     Ex: EA = PC + IR(address)
  2. Indexed Addressing Mode (R = IX, where IX: Index Register)

     Ex: EA = IX + IR(address)
  3. Base Register Addressing Mode

     (R = BAR, where BAR: Base Address Register)

     Ex: EA = BAR + IR(address)

Addressing Modes

6. Direct Address Mode

– Instruction specifies the memory address which can be used directly to access the memory.
– Faster than the other memory addressing modes.
– Too many bits are needed to specify the address for a large physical memory space.
– EA = IR(addr) (IR(addr): address field of IR)

7. Indirect Addressing Mode

– The address field of an instruction specifies the address of a memory location that contains the address of the operand.
– When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits.
– Slow to acquire an operand because of an additional memory access
– EA = M[IR(address)]

Addressing Modes

Example :

PC = 200

R1 = 400

XR = 100

AC

Address	Memory	
200	Load to AC	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

Addressing Mode	Effective Address		Content of AC
Direct address	500	/* AC ← (500) */	800
Immediate operand	-	/* AC ← 500 */	500
Indirect address	800	/* AC ← ((500)) */	300
Relative address	702	/* AC ← (PC+500) */	325
Indexed address	600	/* AC ← (RX+500) */	900
Register	-	/* AC ← R1 */	400
Register indirect	400	/* AC ← (R1) */	700
Autoincrement	400	/* AC ← (R1)+ */	700
Autodecrement	399	/* AC ← -(R) */	450

Data Transfer and Manipulation Instructions

- Computer instructions can be classified into three categories.
    1. Data Transfer Instructions
    2. Data Manipulation Instructions
    3. Program Control Instructions
- Data Transfer Instructions transfer the data from one location to another location.
- Data manipulation instructions performs arithmetic ,logic and shift operations on the data.
- Program control instructions provide decision making and change the path taken by the program when executed in the computer.

Data Transfer Instructions

- Data Transfer instructions move the data between

  Memory ---- Processor register

  Processor register ---- Input/output

  Processor registers ---- Processor registers

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

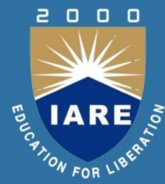
Table : Typical data Transfer Instructions

Data Transfer Instructions

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$
Autodecrement	LD -(R1)	$R1 \leftarrow R1 - 1, AC \leftarrow M[R1]$

Table : Addressing modes for load instruction

Data Manipulation Instructions



- Data Manipulation Instructions are of three basic types.
  1. Arithmetic Instructions
  2. Logical and Bit Manipulation Instructions
  3. Shift Instructions

1. Arithmetic Instructions

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

Mnemonic
ADDI
ADDF
ADDD

Table: Typical Arithmetic Instructions

2. Logical and Bit manipulation Instructions

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

Table : Logical and Bit Manipulation Instructions

Data Manipulation Instructions

Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right thru carry	RORC
Rotate left thru carry	ROLC

Op REG TYPE RL COUNT- ShiftInstruction

Program Control

- When the program control instruction is executed it change the address value in the program counter and cause the flow of control to be altered.

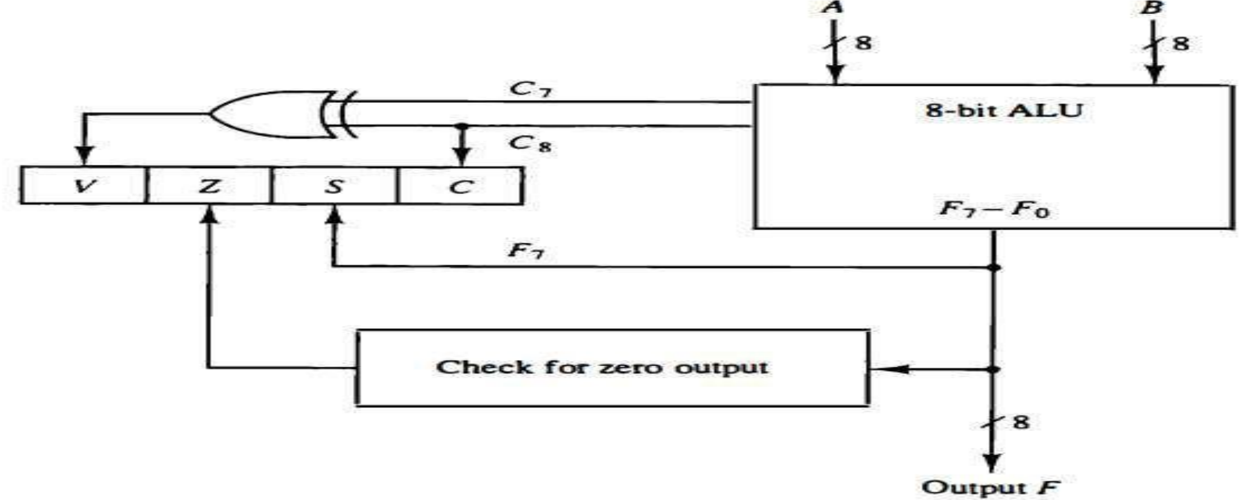
Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by –)	CMP
Test(by AND)	TST

Table: Program Control Instructions

- Branch and Jump instructions are conditional and Unconditional Instructions.
- CMP and Test set some of the bits in PSW(Processor status Word).

Program Control

- In Basic Computer, the processor had several (status) flags – 1 bit value that indicated various information about the processor's state – E, FGI, FGO, I, IEN, R.
- In some processors, flags like these are often combined into a register – the processor status register (PSR); sometimes called a processor status word (PSW).
- Common flags in PSW are
  - C (Carry): Set to 1 if the carry out of the ALU is 1
  - S (Sign): The MSB bit of the ALU's output
  - Z (Zero): Set to 1 if the ALU's output is all 0's
  - V (Overflow): Set to 1 if there is an overflow



Status Flag Circuit

Program Control

- Bit C (carry) is set to 1 if the end carry C8 is 1. It is cleared to 0 if the carry is 0.
- Bit S (sign) is set to 1 if the highest-order bit F, is 1. It is set to 0 if the bit is 0.
- Bit Z (zero) is set to 1 ifthe output ofthe ALU contains all O's. it is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.

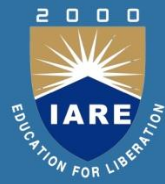
•**Subroutine** Call and Return

A subroutine is a self-contained sequence of instructions that performs a given computational task.

During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program.

Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

Program Control



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

Table: conditional branch instructions

Program Control

- **Types of Interrupts**

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

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

- Internal interrupts are synchronous with the program while external interrupts are asynchronous.
- If the program is rerun, the internal interrupts will occur in the same place each time.
- External interrupts depend on external conditions that are independent of the program being executed at the time.
- External and internal interrupts are initiated from signals that occur in the hardware of the CPU.

Program Control

- A software interrupt is initiated by executing an instruction.
- Software interrupt is a special call instruction that behave 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.

Addition And Subtraction

Addition (subtraction) algorithm: when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different (identical), compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

TABLE: Addition and Subtraction of Signed Magnitude Numbers

Addition And Subtraction

Hardware Implementation:

- Let A and B be two registers that hold the magnitudes of the numbers, and As and Bs be two flip-flops that hold the corresponding signs. The result of the operation may be transferred to a third register: the result is transferred into A and As . Thus A and As together form an accumulator register.
- Consider now the hardware implementation of the algorithms above. First, a parallel-adder is needed to perform the micro operation A + B . Second, a comparator circuit is needed to establish if $A > B$, $A = B$, or $A < B$. Third, two parallel-subtractor circuits are needed to perform the micro operations A - B and B - A. The sign relationship can be determined from an exclusive OR gate with A, and B, as inputs.
- This procedure requires a magnitude comparator, an adder, and two subtractors. First, we know that subtraction can be accomplished by means of complement and add. Second, the result of a comparison can be determined from the end carry after the subtraction.

Addition And Subtraction

- Figure shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A, and B, . Subtraction is done by adding A to the 2' s complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF holds the overflow bit when A and B are added.
- The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M.
- The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in Fig. The M signal is also applied to the input carry of the adder. When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A + B.

Addition And Subtraction

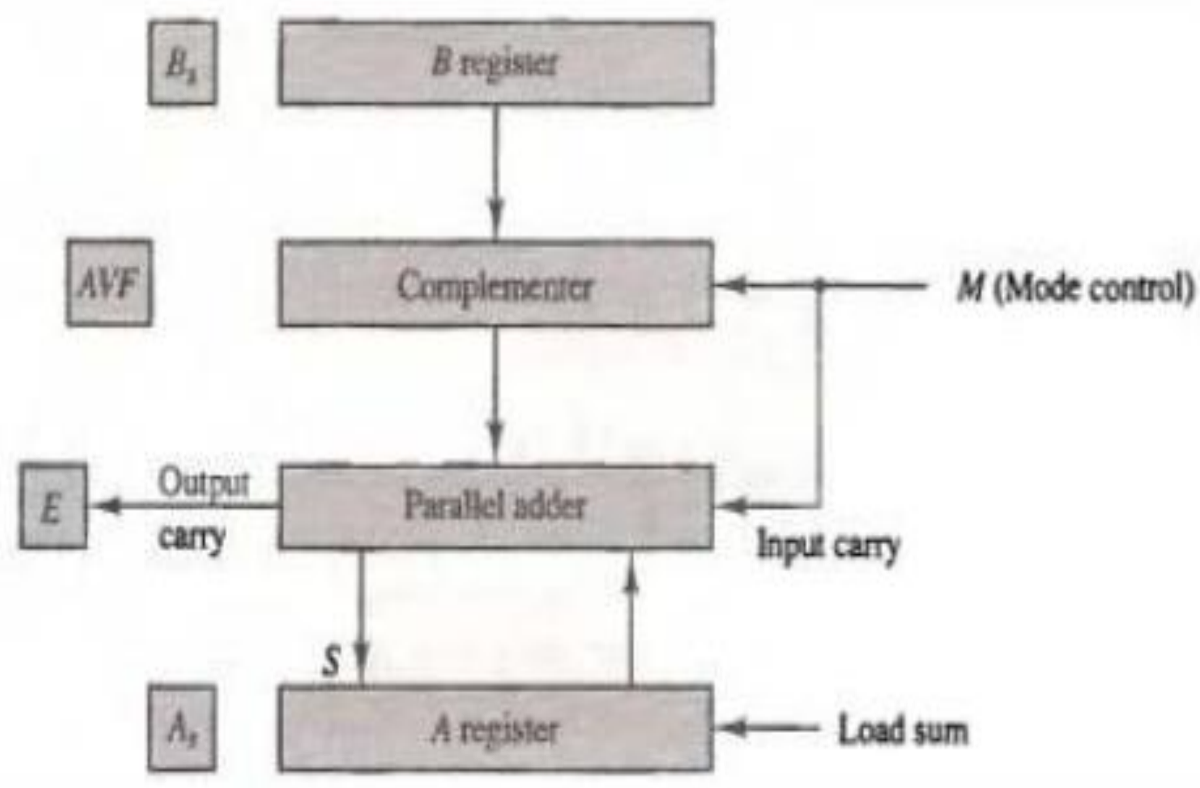


Figure: Hardware for signed magnitude addition and subtraction.

- When M=1, the 1's complement of B is applied to the adder, the input carry is 1, and output S = A + B̀ + 1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A - B.

Addition And Subtraction

- The two signs A, and B are compared by an exclusive-OR gate. If the output of the gate Is 0, the signs are identical if it is 1, the signs are different For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a micro operation EA <-A + B. where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1.
- The value of E is transferred into the add-overflow flip-flop AVF. The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- A 1 in E indicates that A >=B and the number in A is the correct result. If this number is zero, the sign A, must be made positive to avoid a negative zero. A 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one mlcrooperation A <-A+ 1 .

Addition And Subtraction

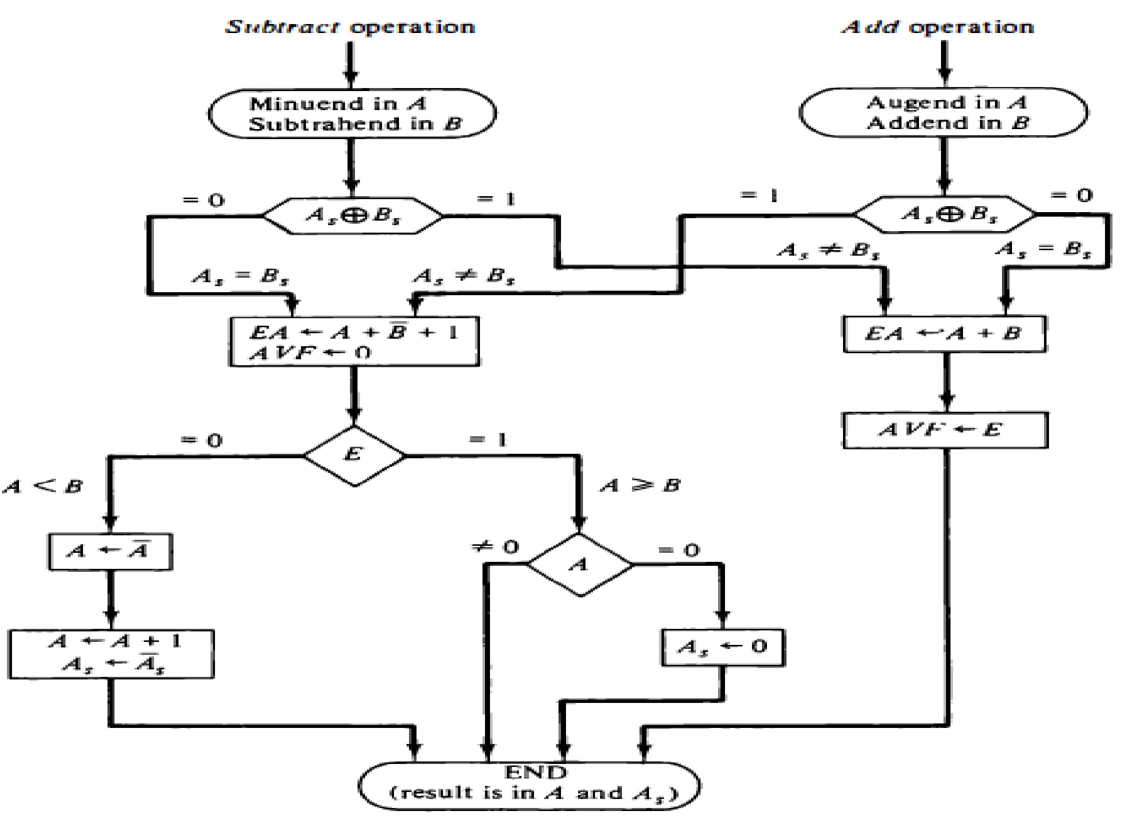


Figure: Flowchart for add and subtract operations.

- However, we assume that the A register has circuits for micro operations complement and increment, so the 2' s complement is obtained from these two micro operations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A, is required. However, when A < B, the sign of the result is the complement of the original sign of As.

Addition and Subtraction with Signed-2'sComplement Data

- The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2' s complement form.
- The algorithm for adding and subtracting two binary numbers in signed- 2' s complement representation is shown in the flowchart of Fig. The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise.
- The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR . Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign.

Addition And Subtraction

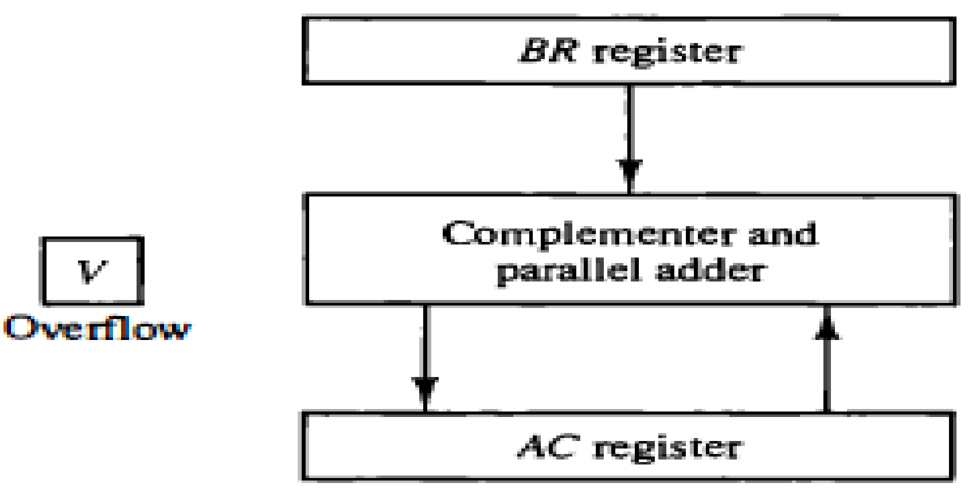


Figure: Hardware for signed2's complement addition and subtraction.

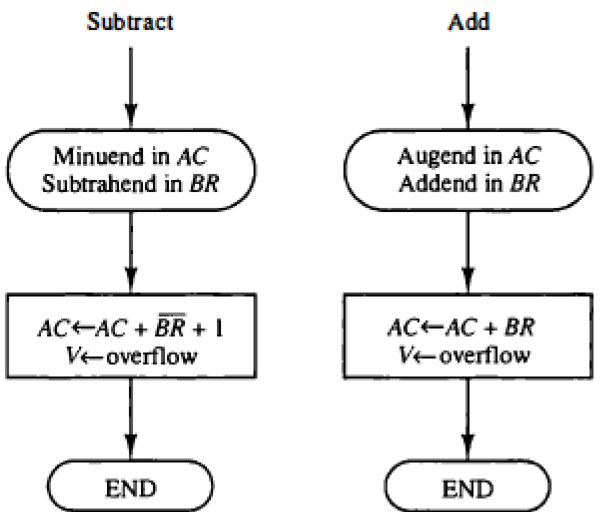


Figure :Algorithm for adding and subtracting numbers in signed 2's complement representation.

Multiplication Algorithms

```
 23         10111    Multiplicand
 19       × 10011    Multiplier
          -------
            10111
           10111
          00000      +
         00000
        10111
        ---------
437     110110101    Product
```

- The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

Multiplication Algorithms

Hardware Implementation for Signed-Magnitude Data

- The hardware for multiplication consists of the equipment shown in Fig. plus two more registers. These registers together with registers A and B are shown in Fig. The multiplier is stored in the Q register and its sign in Qs . The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.
- The multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig.

Multiplication Algorithms

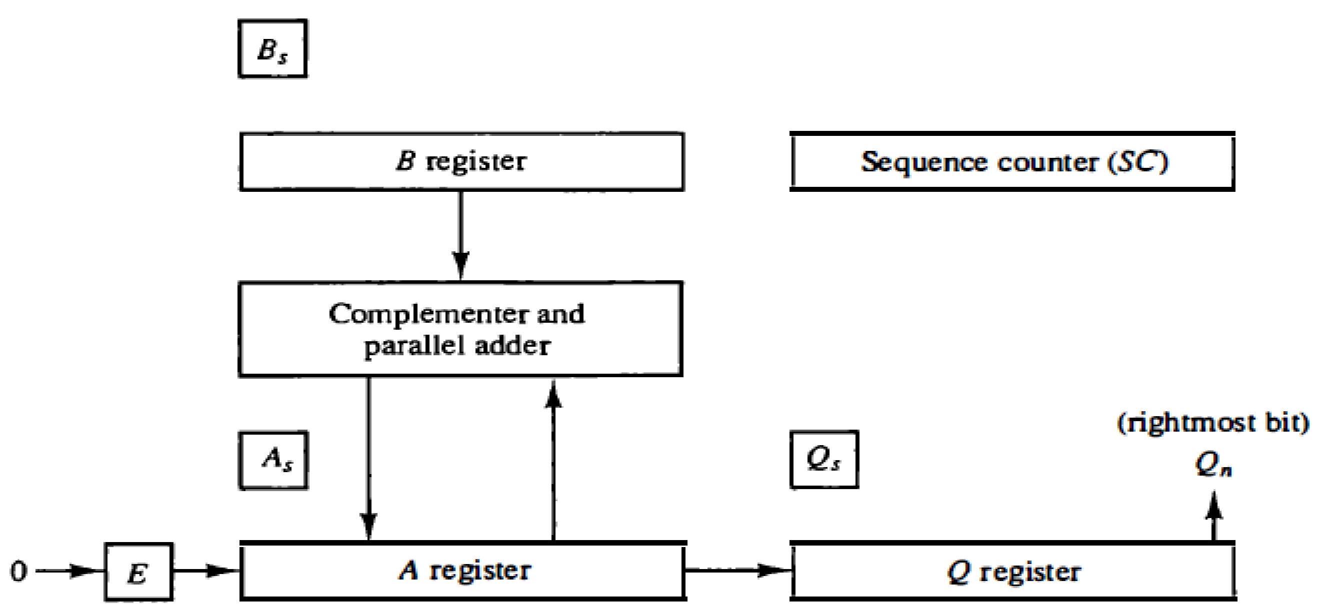


Figure : Hardware for multiply operation.

- The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Qn, will hold the bit of the multiplier, which must be inspected next.

Multiplication Algorithms

Hardware Algorithm

Multiply operation

Multiplicand in B
Multiplier in Q

$A_s \leftarrow Q_s \oplus B_s$
$Q_s \leftarrow Q_s \oplus B_s$
$A \leftarrow 0,\ E \leftarrow 0$
$SC \leftarrow n - 1$

Q_n

$= 0$

$= 1$

$EA \leftarrow A + B$

shr EAQ
$SC \leftarrow SC - 1$

SC

$\neq 0$

$= 0$

END
(product is in AQ)

Figure : Flowchart for multiply operation.

Multiplication Algorithms

- flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs, and Qs, respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.
- After the initialization, the low-order bit of the multiplier in Q, is tested. If it is a 1, the multiplicand in B is added to the present partial product in A . I fit is a 0 , nothing is done.
- Register EAQ is then shifted once to the right to form the new partial product The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0.The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Multiplication Algorithms

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Q_n = 0; shift right EAQ	0	01000	10110	010
Q_n = 0; shift right EAQ	0	00100	01011	001
Q_n = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

TABLE : Numerical Example for Binary Multiplier

Booth Multiplication Algorithm

- For example, the binary number 001 110 (+ 14) has a string of 1's from 2^3 to 21 (k = 3, m = 1). The number can be represented as 2k+ 1 - 2m = 24 - 21 = 16 -2 = 14. Therefore, the multi plication M x 14, where M is the multiplicand and 14 the multiplier, can be done $M \times 2^4 - M \times 2^1$.
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.
- For example, a multiplier equal to - 1 4 is represented in 2's complement as 1 10010 and is treated as - 24 + 22 - 21 = - 14.

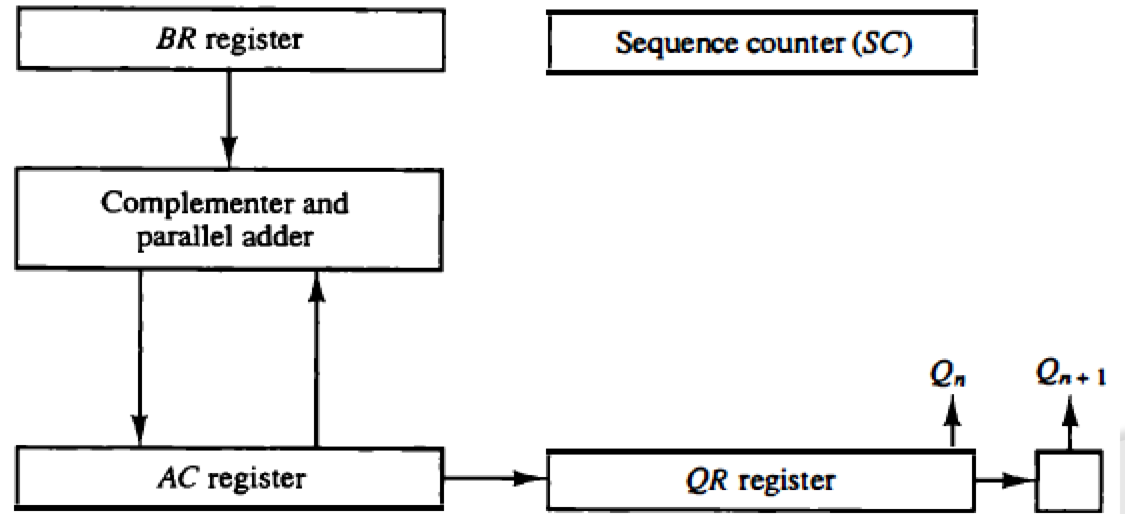


Figure: Hardware for Booth algorithm.

Multiplication Algorithms

- The hardware implementation of Booth algorithm requires the register configuration shown in Fig. This is similar to Fig. except that the sign bits are not separated from the rest of the registers.
- To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q, designates the least significant bit of the multiplier in register QR .
- An extra flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in fig.
- AC and the appended bit Qn+1 are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1 are inspected.

Multiplication Algorithms

Multiply

Multiplicand in BR
Multiplier in QR

$AC \leftarrow 0$
$Q_{n+1} \leftarrow 0$
$SC \leftarrow n$

$Q_n Q_{n+1}$

$= 10$

$= 01$

$= 00$
$= 11$

$AC \leftarrow AC + \overline{BR} + 1$

$AC \leftarrow AC + BR$

ashr (AC & QR)
$SC \leftarrow SC - 1$

SC

$\neq 0$

$= 0$

END

Figure : Booth algorithm for multiplication o f signed 2's complement numbers.

Multiplication Algorithms

- If the two bits are equal to 10, it means that the first 1 in a string of 1' s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC . If the two bits are equal to 01, it means that the first 0 in a string of 0' s has been encountered. This requires the addition of the multiplicand to the partial product in AC .
- When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow.
- The next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

Multiplication Algorithms

- A numerical example of Booth algorithm is shown in Table 10-3 for n = 5. It shows the step-by-step multiplication of (- 9) x (- 13) = + 117. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive. The final value of Qn+1 is the original sign bit of the multiplier and should not be taken as part of the product.

Q_n Q_{n+1}	BR = 10111 $\overline{BR}$ + 1 = 01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

TABLE: Example of Multiplication with Booth Algorithm

Array Multiplier

		b_1	b_0
		a_1	a_0
		a_0b_1	a_0b_0
	a_1b_1	a_1b_0	
c_3	c_2	c_1	c_0

a_0 b_1 b_0

a_1 b_1 b_0

HA C S   HA C S

c_3 c_2 c_1 c_0

Figure : 2-bit by 2-bit array multiplier.

Array Multiplier

- The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0• The first partial product is formed by multiplying a0 by b1 b0• The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces a 0.
- This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b1 b0 and is shifted one position to the left.
- The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

Array Multiplier

Figure : 4-bit by 3-bit array multiplier.

Division Algorithms

Divisor:	11010	Quotient = Q
B = 10001	0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Figure : Example of binary division.

Division Algorithms

Divisor $B = 10001$, $\bar{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure: Example of binary division with digital hardware.

Division Algorithms

- The hardware divide algorithm is shown in the flowchart of below Fig The dividend is in A and Q and the divisor in B . The sign of the result is transferred into Q, to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.
- As in multiplication, we assume that operands are transferred to registers from a memory unit that has words o f n bits. Since a n operand must b e stored with its sign, one bit o f the word will be occupied by the sign and the magnitude will consist of n - 1 bits.
- A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A .
- If A>=B, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A .

Division Algorithms

Divide operation

Dividend in AQ
Divisor in B

$Q_s \leftarrow A_s \oplus B_s$
$SC \leftarrow n - 1$

$EA \leftarrow A + \overline{B} + 1$

E: = 1 ($A \geq B$) → $EA \leftarrow A + B$, $DVF \leftarrow 1$ → END (Divide overflow)

E: = 0 ($A < B$) → $EA \leftarrow A + B$, $DVF \leftarrow 0$ → Divide magnitudes

Divide magnitudes

shl EAQ

E = 0 → $EA \leftarrow A + \overline{B} + 1$

E = 1 → $A \leftarrow A + \overline{B} + 1$

E = 0 ($A < B$) → $EA \leftarrow A + B$

E = 1 ($A \geq B$) → $Q_n \leftarrow 1$

$SC \leftarrow SC - 1$

SC: ≠ 0 → back to shl EAQ; = 0 → END (Quotient is in Q, remainder is in A)

Figure : Flowchart for divide operation.

Floating-Point Arithmetic Operations

- The register organization for floating-point operations is shown in Fig. There are three registers, BR, AC , and QR . Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.
- It is assumed that each floating-point number has a mantissa in signed magnitude representation and a biased exponent. Thus the AC has a mantissa.
- whose sign is in A, and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A1• The bit in this position must be a 1 for the number to be normalized.
- Note that the symbol AC represents the entire register, that is, the concatenation of As, A, and a . Similarly, register BR is subdivided into Bs, B, and b, and Q R into Qs, Q ,and q.

Floating-Point Arithmetic Operations

- A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E . A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity

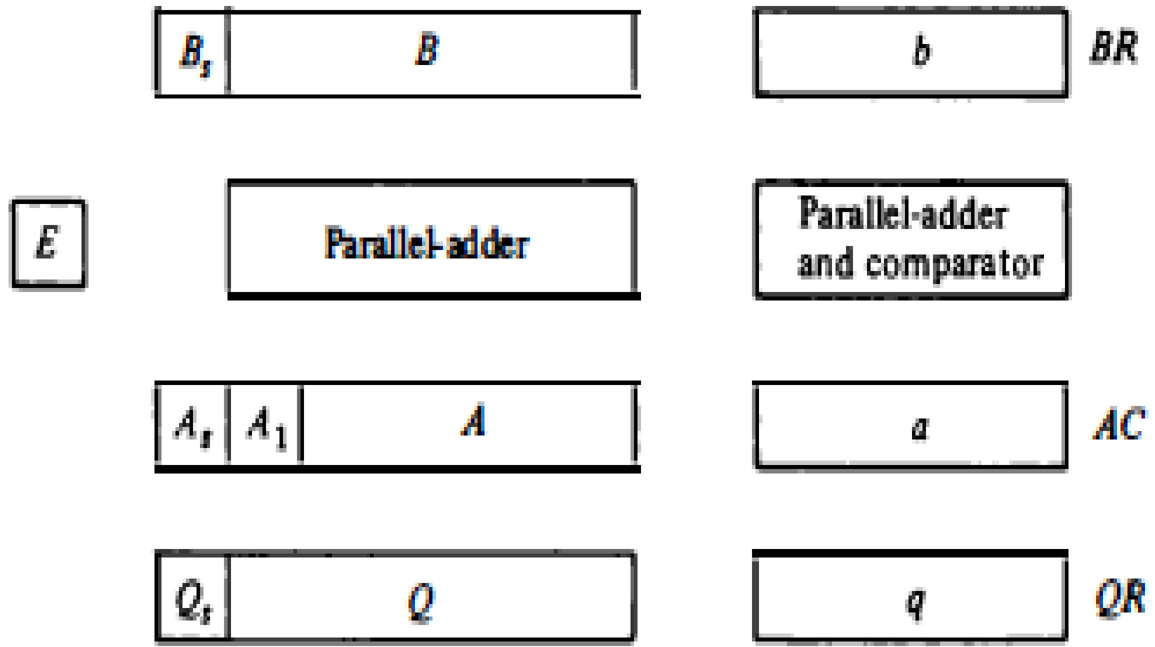


Figure : Registers for floating-point arithmetic operations.

Floating-Point Arithmetic Operations

Addition and Subtraction

- During addition or subtraction, the two floating-point operands are in AC and BR . The sum or difference is formed in the AC . The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result

- The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. If BR is equal to zero, the operation is terminated, with the value in the AC being the result.
- If AC is equal to zero, we transfer the content of BR into AC and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.

Floating-Point Arithmetic Operations

Add or subtract

= 0 BR ≠ 0 AC ≠ 0

= 0

$AC \leftarrow BR$

Check for zeros

Add op Sub

$A_s \leftarrow \bar{A}_s$

$a < b$ $a : b$ $a > b$

$a = b$

shr A, $a \leftarrow a + 1$

shr B, $b \leftarrow b + 1$

Align mantissas

Sub op Add

= 0 $A_s \oplus B_s$ = 1 = 1 $A_s \oplus B_s$ = 0

$EA \leftarrow A + \bar{B} + 1$

$EA \leftarrow A + B$

Mantissa addition or subtraction

$A \leftarrow \bar{A} + 1$, $A_s \leftarrow \bar{A}_s$ = 0 E

= 1

≠ 0 A = 0

= 0 A_1 = 1

$AC \leftarrow 0$

= 0 E = 1

shl A, $a \leftarrow a - 1$

shr A, $A_1 \leftarrow E$, $a \leftarrow a + 1$

Normalization

END

Figure : Addition and subtraction of floating-point numbers.

Floating-Point Arithmetic Operations

- The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.
- The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm presented in Fig. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas.
- If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E is equal to 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number.

Floating-Point Arithmetic Operations

Multiplication

- The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents.

The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product

- Steps 2 and 3 can b e done simultaneously if separate adders are available for the mantissas and exponents.
- The flowchart for floating-point multiplication is shown in Fig. The two operands are checked to determine if they contain a zero.
- If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

Floating-Point Arithmetic Operations

Multiply

Multiplicand in BR
Multiplier in QR

BR: $= 0$ / $\neq 0$

QR: $= 0$ / $\neq 0$

$AC \leftarrow 0$

$a \leftarrow q$

$a \leftarrow a + b$

$a \leftarrow a - \text{bias}$

Multiply mantissa as in Fig. 10-6

A_1: $= 0$ / $= 1$

shl AQ
$a \leftarrow a - 1$

END
(product is in AC)

Figure : Multiplication of floating-point numbers.

Division

The division algorithm can be subdivided into five parts:

1. Check for zeros.
2. Initialize registers and evaluate the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

- The flowchart for floating-point division is shown in Fig. The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message.
- An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.

Floating-Point Arithmetic Operations

Divisor in BR
Dividend in AC

BR: $= 0$ → Divide by zero; $\neq 0$ → continue

AC: $= 0$ → $QR \leftarrow 0$; $\neq 0$ → continue

$Q_s \leftarrow A_s \oplus B_s$
$Q \leftarrow 0$
$SC \leftarrow n - 1$

$EA \leftarrow A + \overline{B} + 1$

E: $= 1$ ($A \geq B$) → $A \leftarrow A + B$, shr A, $a \leftarrow a + 1$; $= 0$ ($A < B$) → $A \leftarrow A + B$

$a \leftarrow a + \overline{b} + 1$

$a \leftarrow a +$ bias

$q \leftarrow a$

Divide magnitude of mantissas as in Fig. 10-13

END
(Quotient is in QR)

Figure : Division of floating-point numbers.

BCD Adder

- Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder.
- The adder will form the sum in binary and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols K, Z8, Z4, Z2, and Z1• K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.
- The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the second column of the table.

Floating-Point Arithmetic Operations

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

- When the binary sum is greater than 1001, we obtain a non valid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required.

Floating-Point Arithmetic Operations

- The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry K = 1. The other six combinations from 1010 to 1 1 1 1 that need a correction have a 1 in position Z8•
- To distinguish them from binary 1000 and 1001 which also have a 1 in position Z8 we specify further that either Z4 or z, must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function C = K + Z8 Z4 + Z8 Z2. When C = 1, it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage.
- A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum.

Floating-Point Arithmetic Operations

- When the output-carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal2

Addend
Augend
4-bit binary adder
Carry out
K
Carry in
Z_8 Z_4 Z_2 Z_1
Output carry
"0"
4-bit binary adder
S_8 S_4 S_2 S_1

Figure : Block diagram of BCD adder.

Decimal Arithmetic Operations

Addition and Subtraction

- The algorithms for arithmetic operations with decimal data are similar to the algorithms for the corresponding operations with binary data.
- The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitude numbers provided that we interpret the micro operation symbols in the proper manner. Similarly, the algorithm for binary signed-2's complement numbers applies to decimal signed-10's complement numbers. The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers.

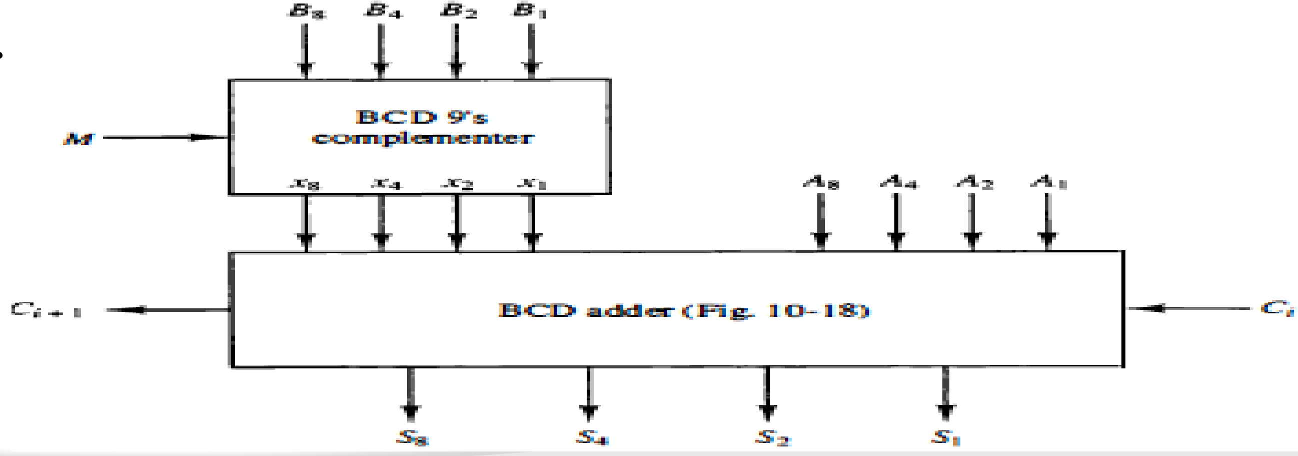
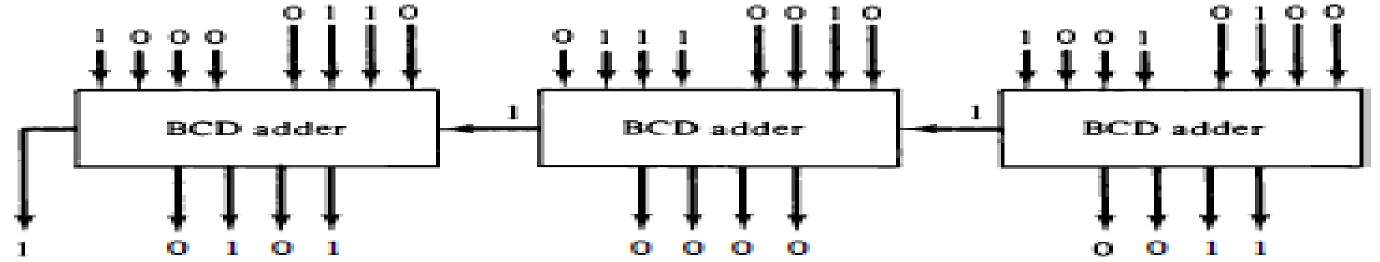


Figure : One stage of a decimal arithmetic unit.

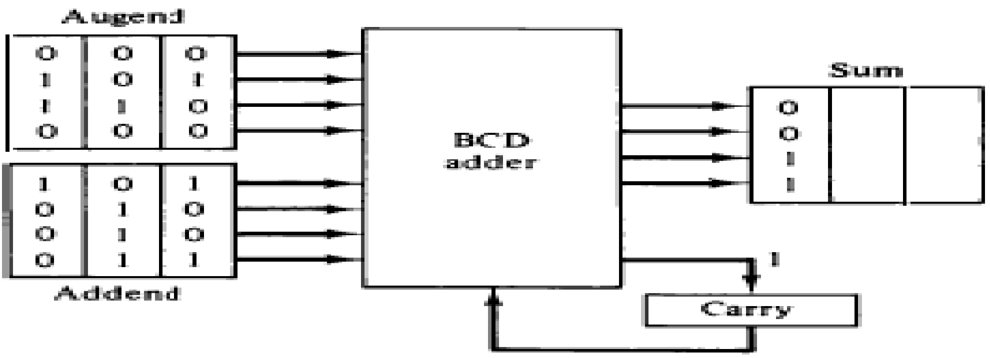
Decimal Arithmetic Operations

- Decimal data can be added in three different ways, as shown in The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one rnicrooperation.
- In the digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel.
- The sum is formed by shifting the decimal numbers through the BCD adder one at a time. For k decimal digits, this configuration requires k microoperations, one for each decimal shift. In the all serial adder, the bits are shifted one at a time through a full-adder.
- The binary sum formed after four shifts must be corrected into a valid BCD digit. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits.

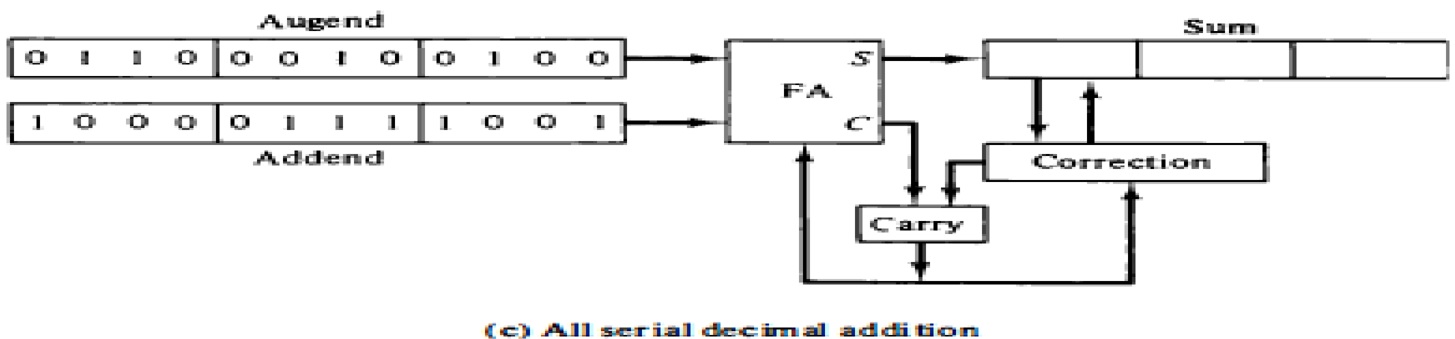
Decimal Arithmetic Operations



(a) Parallel decimal addition: 624 + 879 = 1503



(b) Digit-serial, bit-parallel decimal addition



(c) All serial decimal addition

Figure: Three ways of adding decimal numbers.

Multiplication

- The registers organization for the decimal multiplication is shown in Fig. We are assuming here four-digit numbers, with each digit occupying four bits, for a total of 16 bits for each number. There are three registers, A, B, and Q, each having a corresponding sign flip-flop AS, BS, and QS, .
- Registers A and B have four more bits designated by A, and B, that provide an extension of one more digit to the registers. The BCD arithmetic unit adds the five digits in parallel and places the sum in the five-digit A register. The end-carry goes to flip-flop E.
- The purpose of digit A, is to accommodate an overflow while adding the multiplicand to the partial product during multiplication. The purpose of digit B, is to form the 9's complement of the divisor when subtracted from the partial remainder during the division operation. The least significant digit in register Q is denoted by QL . This digit can be incremented or decremented.

Decimal Arithmetic Operations

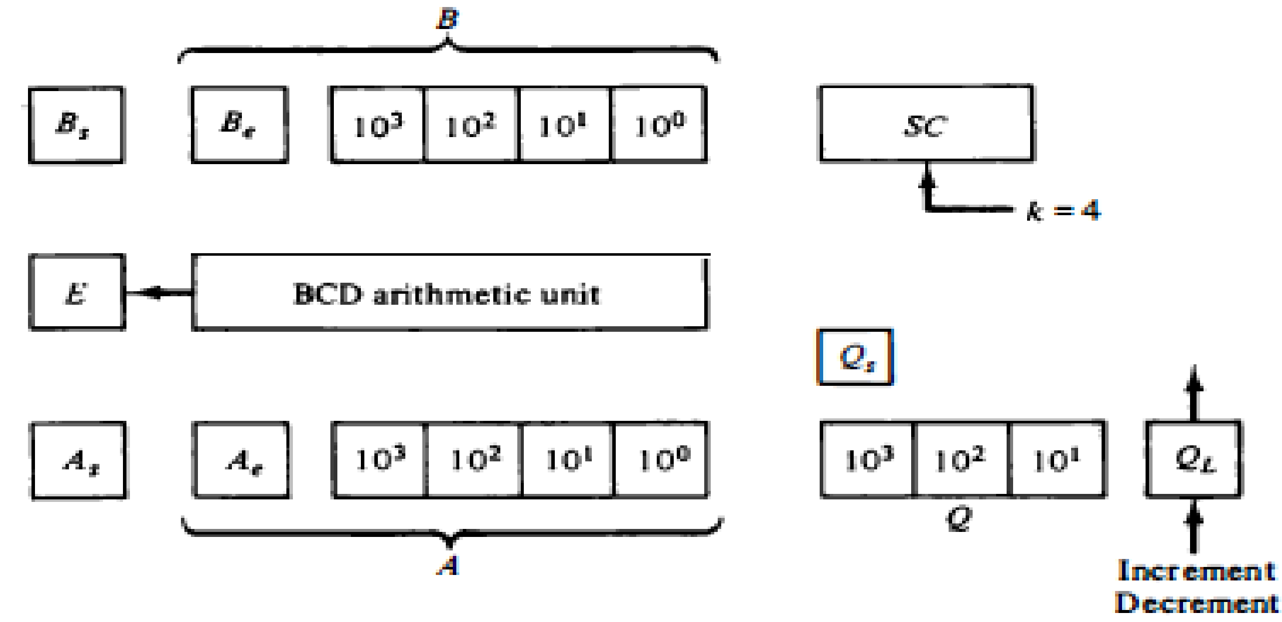


Figure : Registers for decimal arithmetic multiplication and division.

- The decimal multiplication algorithm is shown in Fig. Initially, the entire A register and B, are cleared and the sequence counter SC is set to a number k equal to the number of digits in the multiplier.
- The low-order digit of the multiplier in Q, is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once and QL is decremented.

Decimal Arithmetic Operations

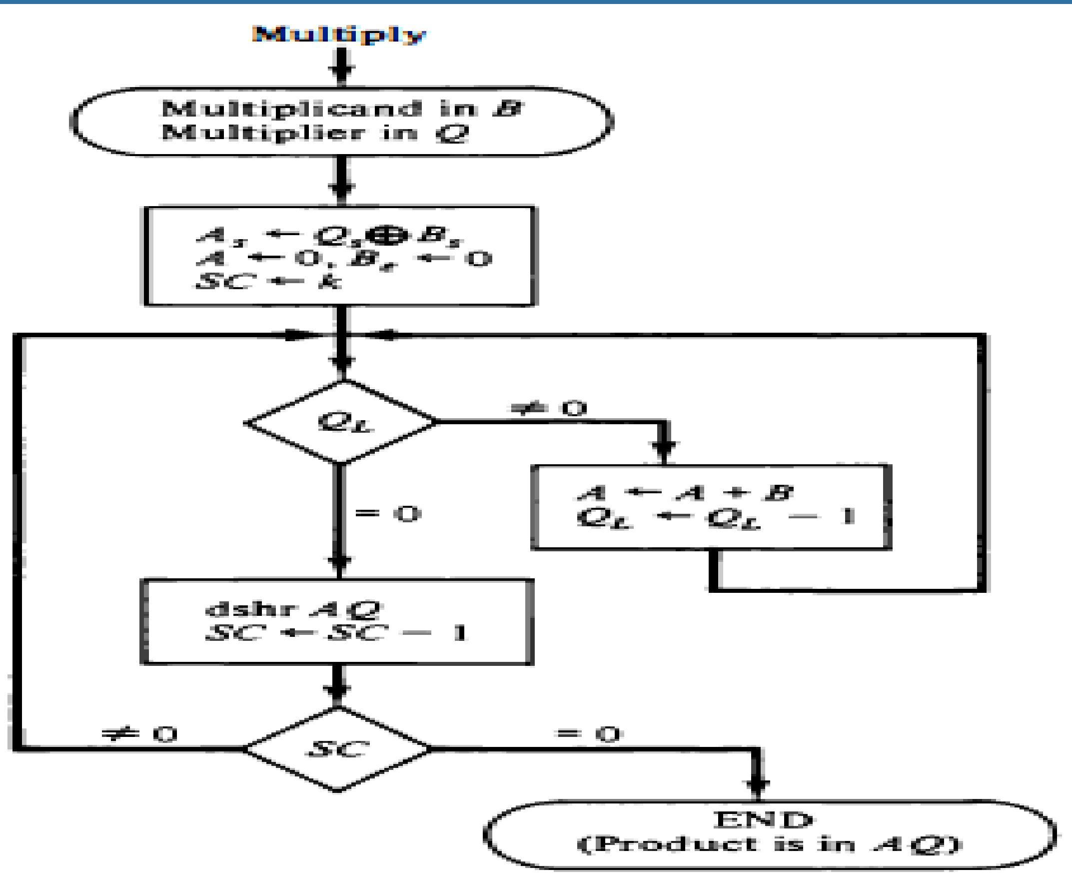


Figure : Flowchart for decimal multiplication.

- QL is checked again and the process is repeated until it is equal to 0. In this way, the multiplicand in B is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in A, and can range in value from 0 to 9.

Decimal Arithmetic Operations

Division

- The decimal division algorithm is shown in Fig. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in A.
- The divisor is then subtracted by adding its 10' s complement value. Since Be is initially cleared, its complement value is 9 as required. The carry in E determines the relative magnitude of A and B. If E = 0, it signifies.
- That A < B. In this case the divisor is added to restore the partial remainder and QL stays at 0 (inserted there during the shift). If E = 1, it signifies that A >= B. The quotient digit in QL is incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by E being 0.

Decimal Arithmetic Operations

Divide

Divisor in B
Dividend in AQ

Check for overflow

$Q_s \leftarrow A_s \oplus B_s$
$SC \leftarrow k,\ B_e \leftarrow 0$

dshl AQ

$EA \leftarrow A + \overline{B} + 1$

E: $= 0$, $A < B$; $= 1$, $A \geq B$

$Q_L \leftarrow Q_L + 1$

$EA \leftarrow A + \overline{B} + 1$

E: $= 0$; $= 1$

$A \leftarrow A + B$

$SC \leftarrow SC - 1$

SC: $\neq 0$; $= 0$

END
(Quotient is in Q)
(Remainder is in A)

Figure : Flowchart for decimal division.

MODULE –IV

INPUT-OUTPUT ORGANIZATION AND MEMORY ORGANIZATION

Contents

Memory organization:

- Memory hierarchy
- Main memory
- Auxiliary memory
- Associative memory
- Cache memory
- Virtual memory

Input or output organization:

- Input or output Interface
- Asynchronous data transfer
- Modes of transfer
- Priority interrupt
- Direct memory access.

Course Outcomes

CO 1	Describe Memory hierarchy, main memory and auxiliary memory.
CO 2	Classify the associative memory, cache memory and virtual memory.
CO 3	Describe the Input or output Interface and asynchronous data transfer.
CO 4	Classify the different modes of transfer in Memory organization.
CO 5	Explore the different priority interrupts and direct memory access.

Memory Hierarchy

- At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor.
- When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- A special very-high speed memory called a cache sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate.

Memory Hierarchy

- The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.

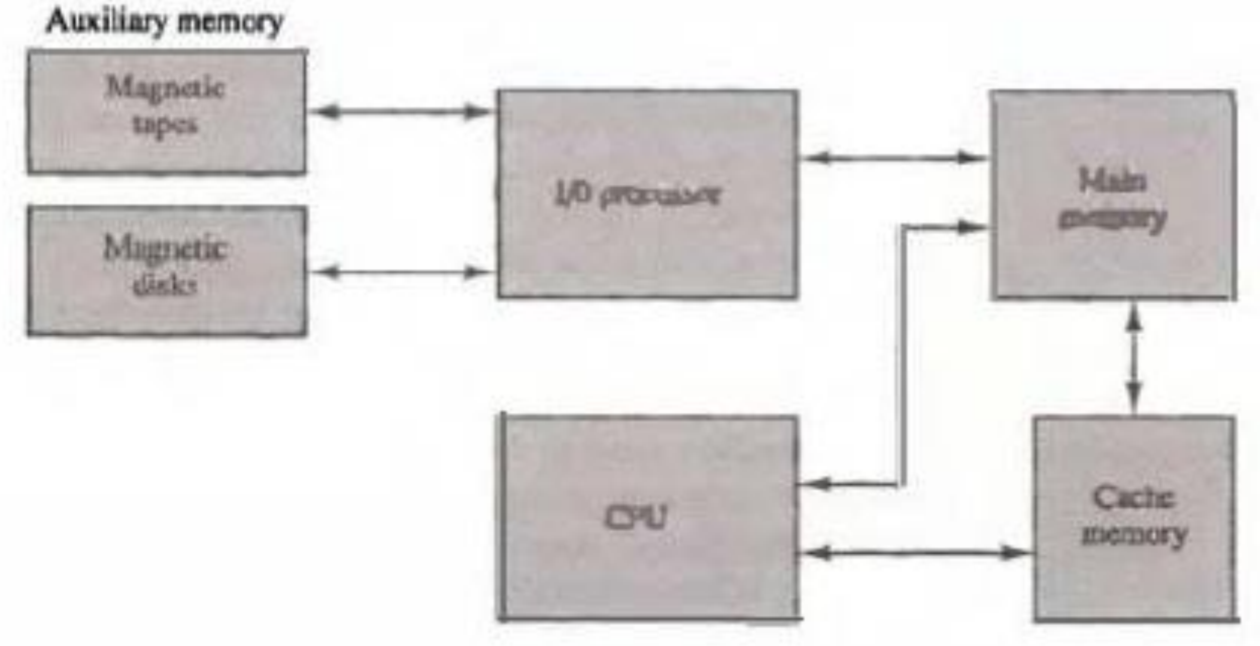


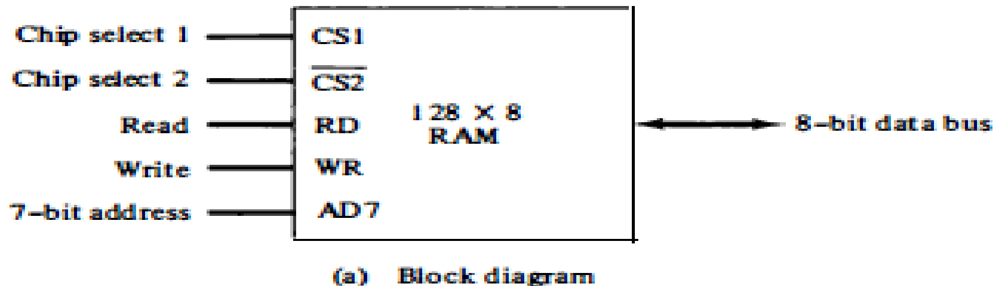
Figure : Memory hierarchy in a computer system.

Main Memory

- The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, static and dynamic.
- The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors.
- The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program.

RAM and ROM Chips:

- A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation, or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers.



(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

Figure: Typical RAM chip.

Main Memory

- The block diagram of a RAM chip is shown in Fig.The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor.
- The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.
- The function table listed in Fig.(b) specifies the operation of the RAM chip. The unit is in operation only when CSI = 1 and CS2 = 0. The bar on top of the second select variable indicates that this input in enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When SC1 = 1 and CS2 = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.

Main Memory

- When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

Memory Address Map:

- a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. For the same-size chip, it is possible to have more bits of ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.
- The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and CS2 = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read.

Main Memory

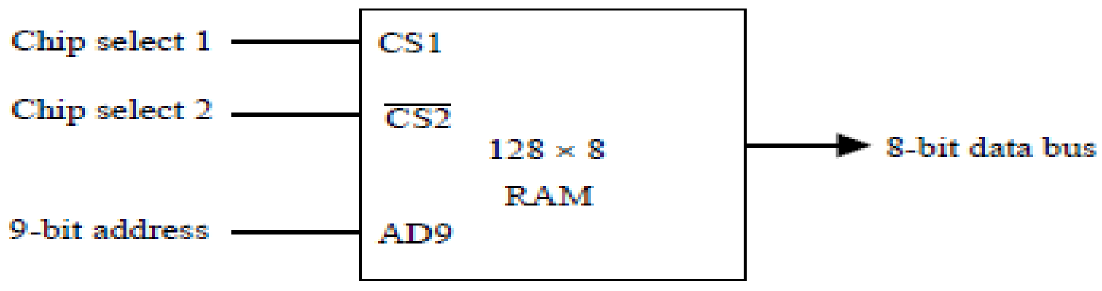


Figure: Typical ROM chip.

- To demonstrate with a particular example, assume that a computer system needs 512
- bytes of RAM and 512 bytes of ROM. The RAM and ROM chips

Component	Hexadecimal address	Address bus 10	9	8	7	6	5	4	3	2	1
RAM 1	0000-007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080-00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100-017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180-01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200-03FF	1	x	x	x	x	x	x	x	x	x

TABLE: Memory Address Map for Microprocomputer

Main Memory

Memory Connection to CPU

- RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in Fig.
- This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 12-1. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus .
- This is done through a 2 x 4 decoder whose outputs go to the CS1 inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

Main Memory

- The selection between RAM and ROM is achieved through bus line 10.The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1 . The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation.
- Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions .

Main Memory

CPU
Address bus
16 – 11 10 9 8 7 – 1 RD WR Data bus
Decoder
3 2 1 0

CS1
$\overline{CS2}$
RD
WR
AD7
128 × 8
RAM 1
Data

CS1
$\overline{CS2}$
RD
WR
AD7
128 × 8
RAM 2
Data

CS1
$\overline{CS2}$
RD
WR
AD7
128 × 8
RAM 3
Data

CS1
$\overline{CS2}$
RD
WR
AD7
128 × 8
RAM 4
Data

CS1
$\overline{CS2}$
1 – 7
8
9
AD9
128 × 8
ROM
Data

Figure: Memory connection to the CPU.

Auxiliary Memory

- The most common auxiliary memory devices used in computer systems are magnetic disks and tapes.

Magnetic Disks:

- A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface.
- Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors.

Sector
Tracks
Read/write head

Figure: Magnetic disk.

Magnetic Tape:

- A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks.
- Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.

Associative Memory

- A memory unit accessed by content is called an associative memory or content addressable memory (CAM). An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.
- **Hardware Organization:**
- It consists of a memory array and logic for m words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.

Associative Memory

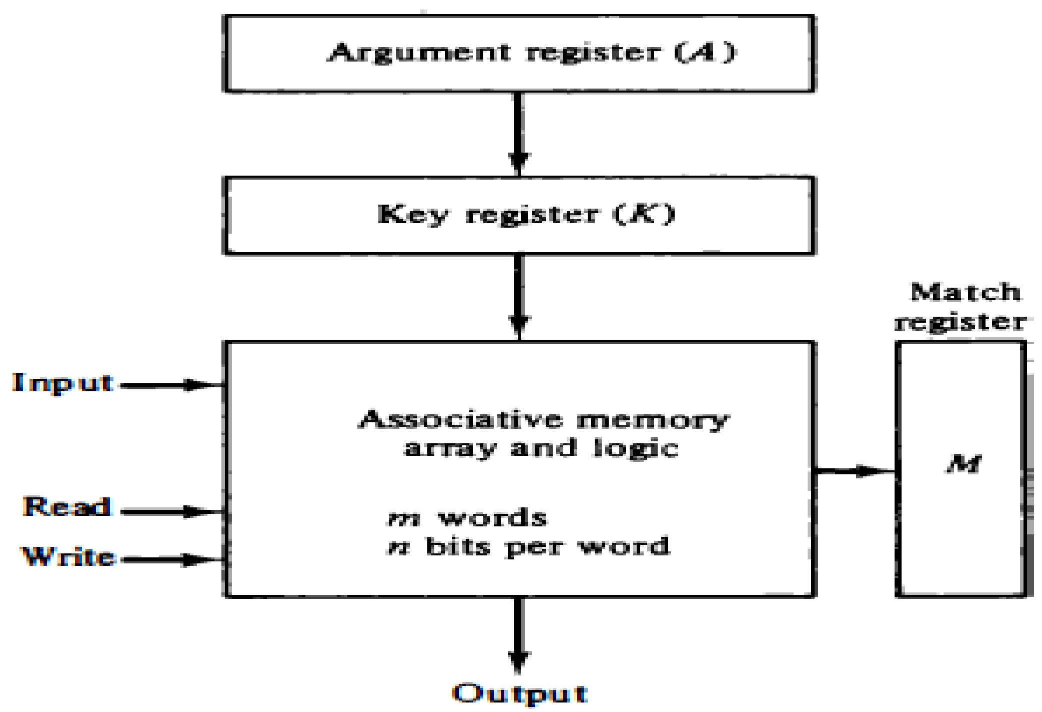


Figure: Block diagram of associative memory.

- The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all l' s. Otherwise, only those bits in the argument that have l's in their corresponding position of the key register are compared.

Associative Memory

- a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

- Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.
- The relation between the memory array and external registers in an associative memory is shown in Fig.
- The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word.

Associative Memory

- Thus cell Cij is the cell for bit j in word i. A bit Ai in the argument register is compared with all the bits in column j of the array provided that Ki = 1. This is done for all columns j = 1, 2, . . . , n.
- If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M1 in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M1 is cleared to 0.

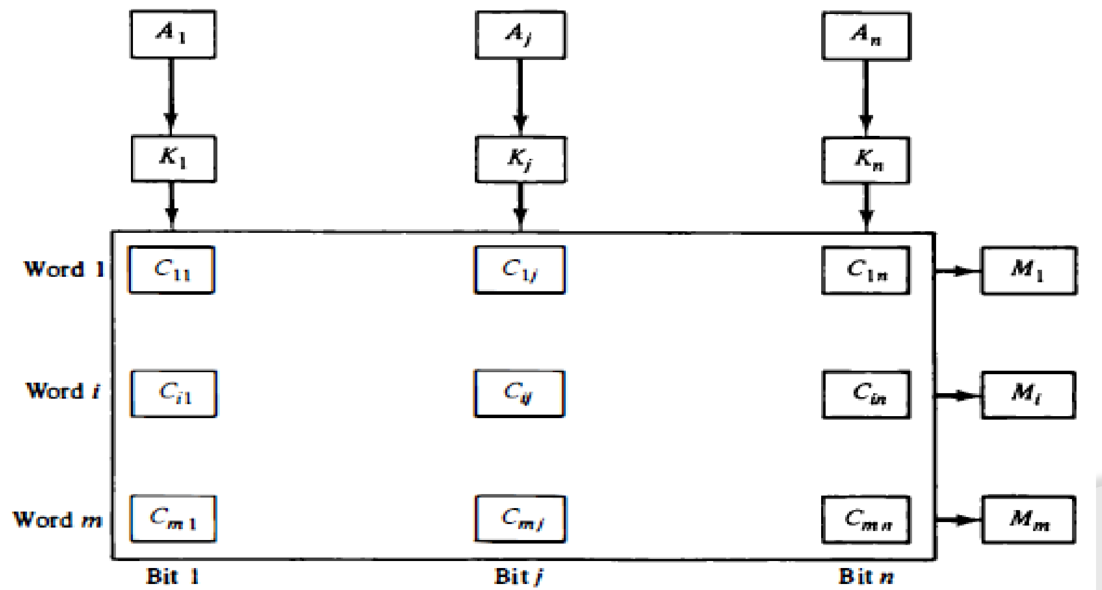


Figure: Associative memory of m word, n cells per word.

Associative Memory

- The internal organization of a typical cell Cij is shown in Fig. It consists of a flip-flop storage element Fi and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation.
- The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in Mi.

Match Logic

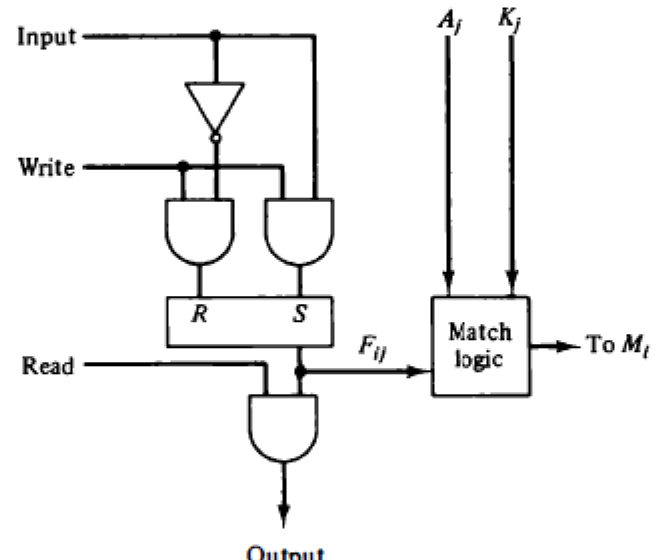


Figure: One cell of associative memory.

Associative Memory

- The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if Aj = Fij for j = 1, 2, . . . , n . Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function.

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

- where xj = 1 if the pair of bits in position j are equal; otherwise, xj = 0. For a word i to be equal to the argument in A we must have all xi variables equal to 1. This is the condition for setting the corresponding match bit M, to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \cdots x_n$$

- and constitutes the AND operation of all pairs of matched bits in a word.

Cache Memory

- The active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory as illustrated in Fig.
- The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.
- The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache.

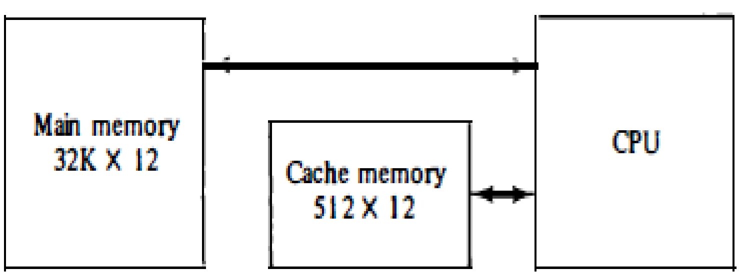


Figure: Example of cache memory.

Cache Memory

- The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory.
- The performance of cache memory is frequently measured in terms of a quantity called hit ratio . When the CPU refers to memory and finds the word in cache, it is said to produce a hit . If the word is not found in cache, it is in main memory and it counts as a miss .
- The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio.
- For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

Cache Memory

- The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

Associative Mapping

- The fastest and most flexible cache organization uses an associative memory. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache.
- The address value of 15 bits is shown as a five-digit octal number and its corresponding 12 -bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address.

Cache Memory

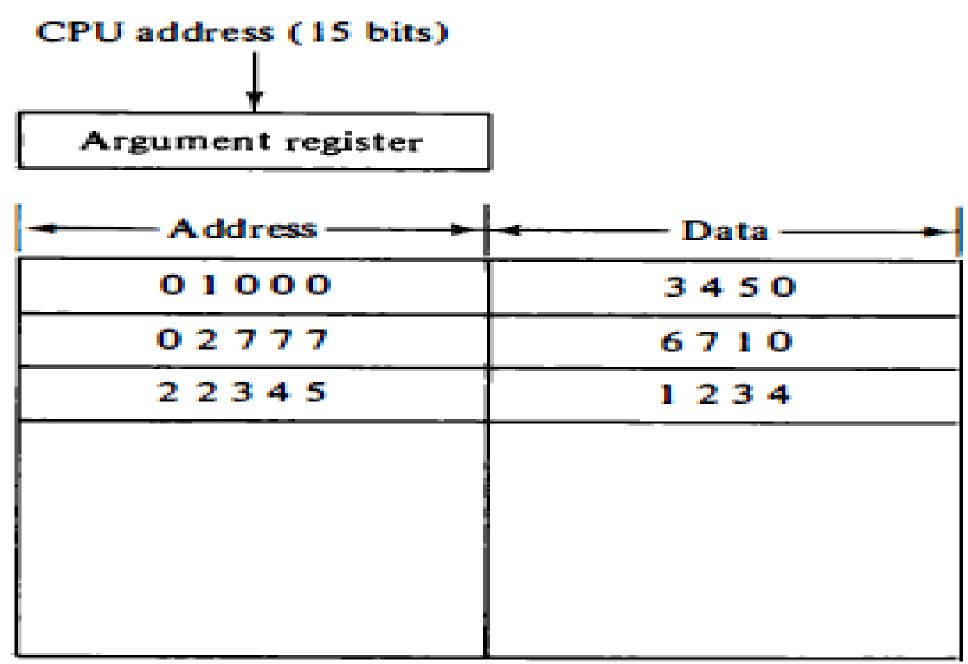


Figure: Associative mapping cache (all numbers in octal).

- If the address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory.
- If the cache is full, an address--data pair must be displaced to make room for a pair that is needed and not presently in the cache.

Cache Memory

Direct Mapping:

- Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits form the tag field.
- The figure shows that main memory needs an address that includes both the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

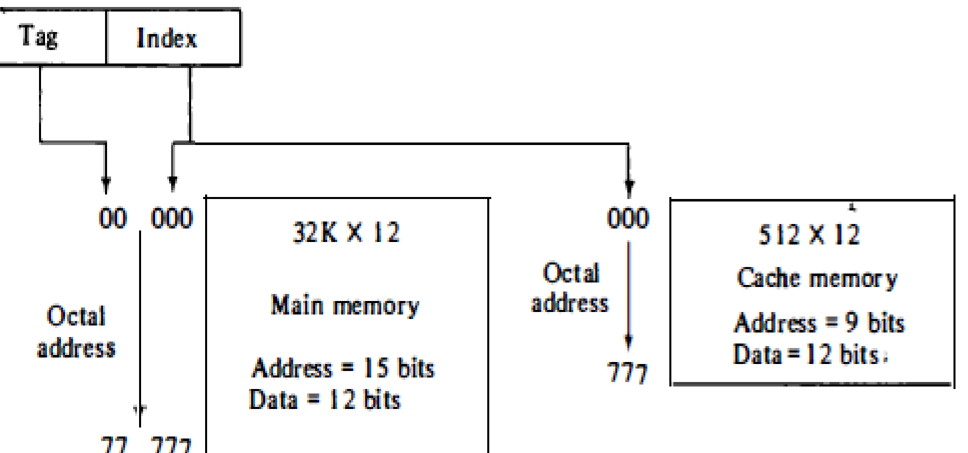
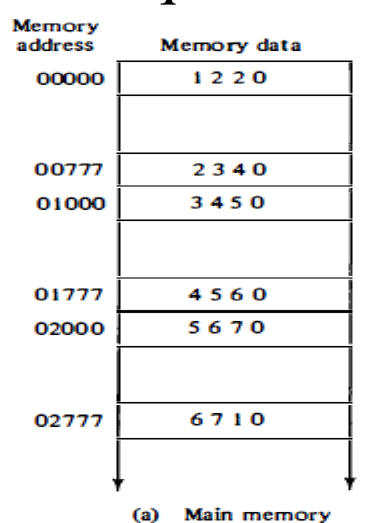


Figure: Addressing relationships between main and cache memories.

Cache Memory

- In the general case, there are 2k words in cache memory and 2n words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n - k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache.
- Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.



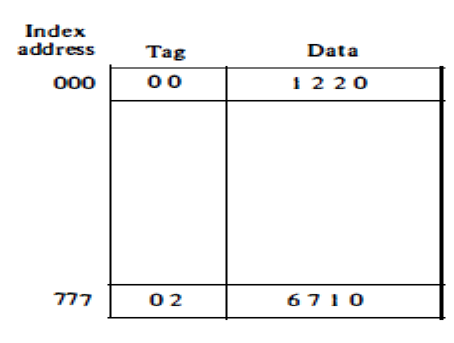


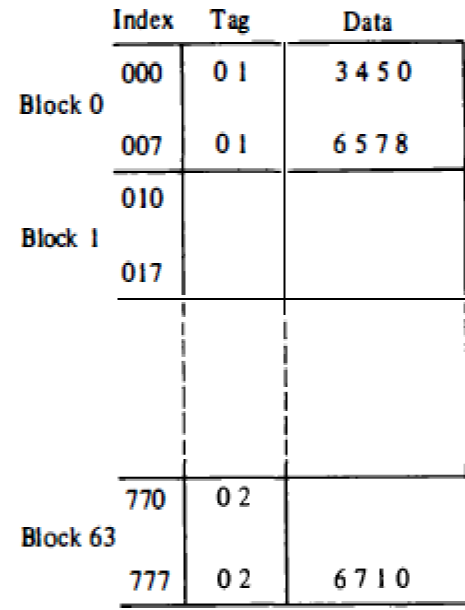
Figure: Direct mapping cache organization.

Cache Memory

- The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly.
- To see how the direct-mapping organization operates, consider the numerical example shown in Fig. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared.
- The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

Cache Memory

- The index field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 blocks of 8 words each, since 64 x 8 = 512. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field.
- The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory.



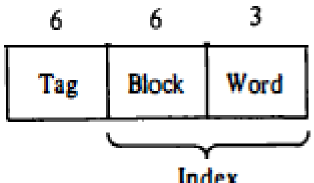


Figure: Direct mapping cache with block size of 8 words.

Set-Associative Mapping:

- It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.
- A third type of cache organization, called set-associative mapping, is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set.

Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

Figure: Two-way set associative mapping cache.

Cache Memory

- The octal numbers listed in Fig. are with reference to the main memory contents illustrated in Fig.(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777.
- When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs.
- thus the name "set-associative." The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache.

Cache Memory

write-through:

- The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the write-through method. This method has the advantage that main memory always contains the same data as the cache.

write-back:

- The second procedure is called the write-back method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory.

Virtual Memory

- Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory.
- Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations.

Virtual Memory

Address Space And Memory Space

- An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in main memory is called a location or physical address. The set of such locations is called the memory space.
- Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing.
- As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 215.
- Suppose that the computer has available auxiliary memory for storing 220 = 1024K words. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

Virtual Memory

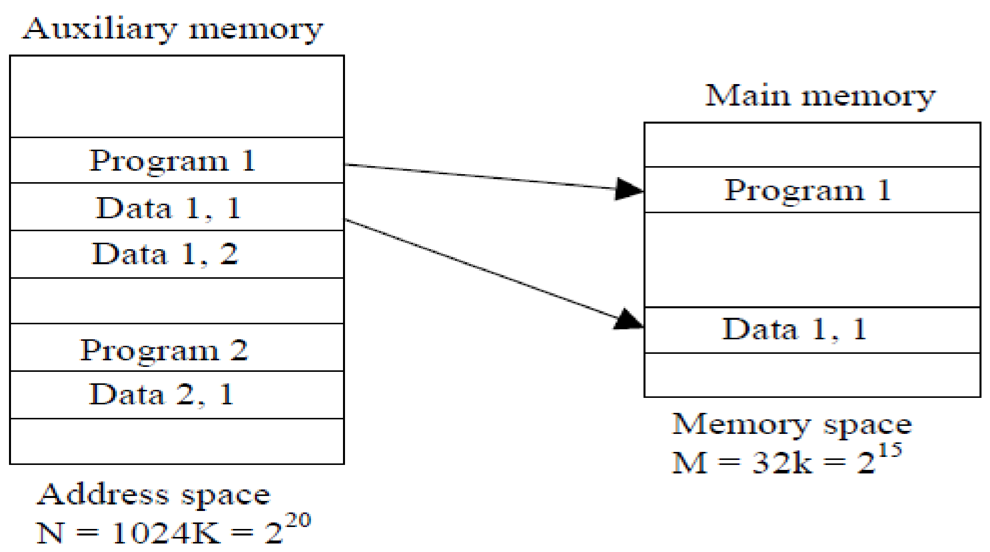


Figure : Relation between address and memory space in a virtual memory system.

- In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses.
- In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address.

Virtual Memory

- To map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU. The mapping table may be stored in a separate memory as shown in Fig.in main memory.

Figure: Memory table for mapping a virtual address.

Address Mapping Using Pages

- Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.
- The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number.
- In the example of Fig. a virtual address has 13 bits. Since each page consists of 210 = 1024 words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

Virtual Memory

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7

Address space
$N = 8K = 2^{13}$

Block 0
Block 1
Block 2
Block 3

Memory space
$M = 4K = 2^{12}$

Figure: Address space and memory space split into groups of 1K words.

- The organization of the memory mapping table in a paged system is shown in Fig. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5 and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively.
- A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table.

Virtual Memory

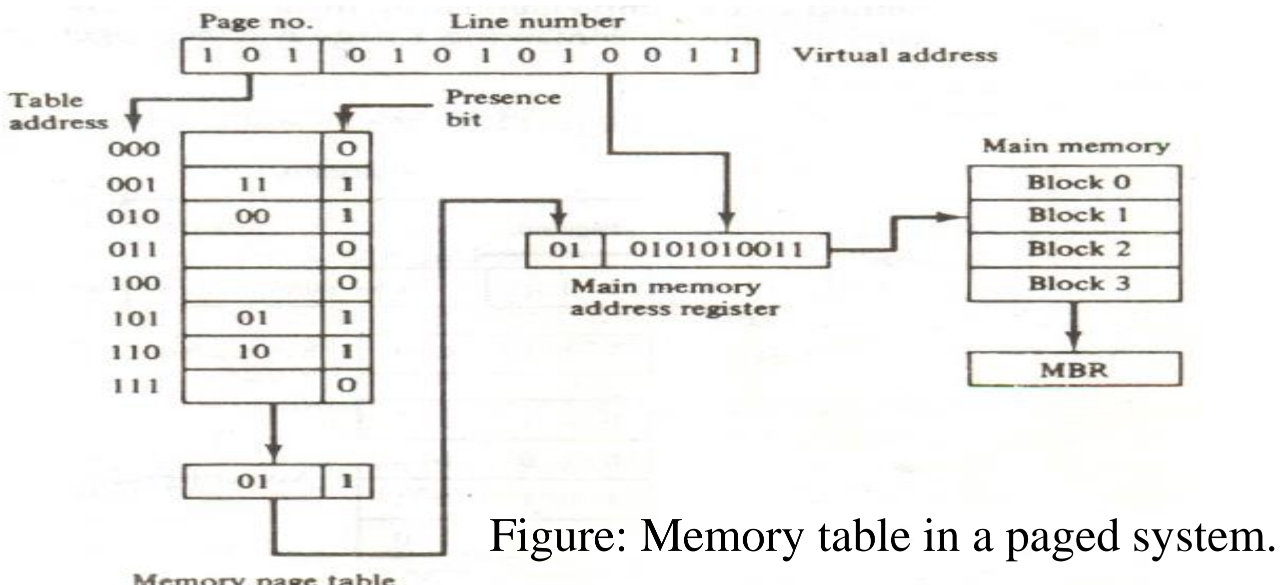


Figure: Memory table in a paged system.

- page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low order bits of the memory address register.
- A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory.

Associative Memory Page Table

The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Consider again the case of eight pages and four blocks as in the example of Fig. We replace the random access memory-page table with an associative memory of four words as shown in Fig. Each entry in the associative memory array consists of two fields. The first three bits specify a field fro storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

Virtual address

Page no.	
1 0 1	Line number

Argument register

1 1 1	0 0

Key register

Page no.	Block no.
0 0 1	1 1
0 1 0	0 0
1 0 1	0 1
1 1 0	1 0

Associative memory

Figure: An associative memory page table.

Input-output Interface

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

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be need.

3. Data codes and formats in peripherals differ form the word format in the CPU and memory.

4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

Input-output Interface

I/O Bus And Interface Modules

- A typical communication link between the processor and several peripherals is shown in Fig. The I/O bus consists of data lines, address lines, and control lines.
- The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit.
- Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller.
- It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device.

Input-output Interface

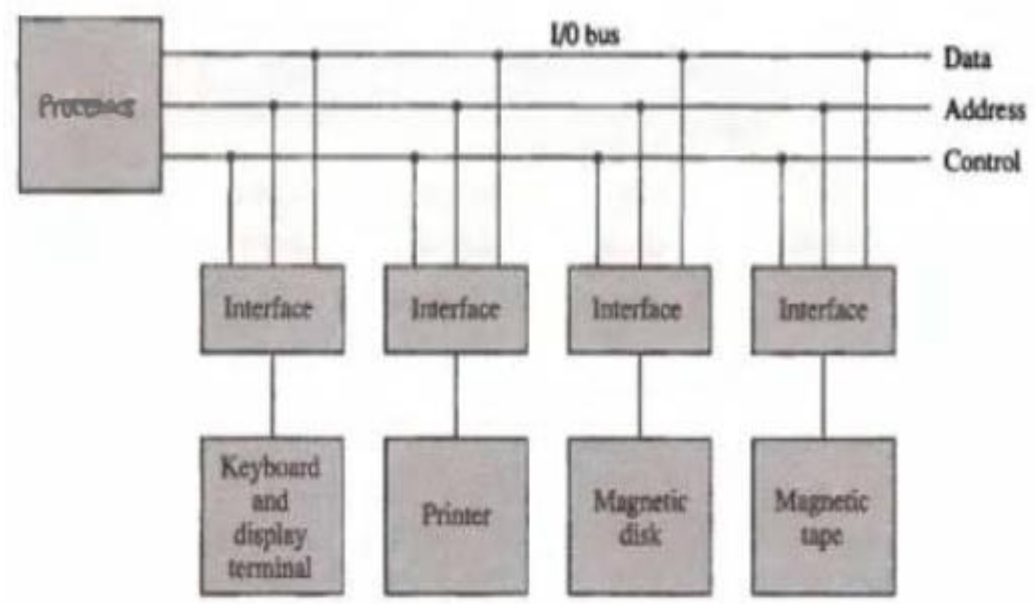


Figure: Connection of I/O bus to input devices.

- The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines.
- Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls.

I/O Versus Memory Bus

- In addition to communicating 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 three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

- In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU).

Input-output Interface

- The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines.
- The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory.

Input-output Interface

Isolated Versus Memory-mapped I/O

- In the **isolated I/O** configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line.
- This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.
- The isolated I/O method isolates memory word and not for an I/O addresses. The other alternative is to use the same address space for both memory and I/O.

Input-output Interface

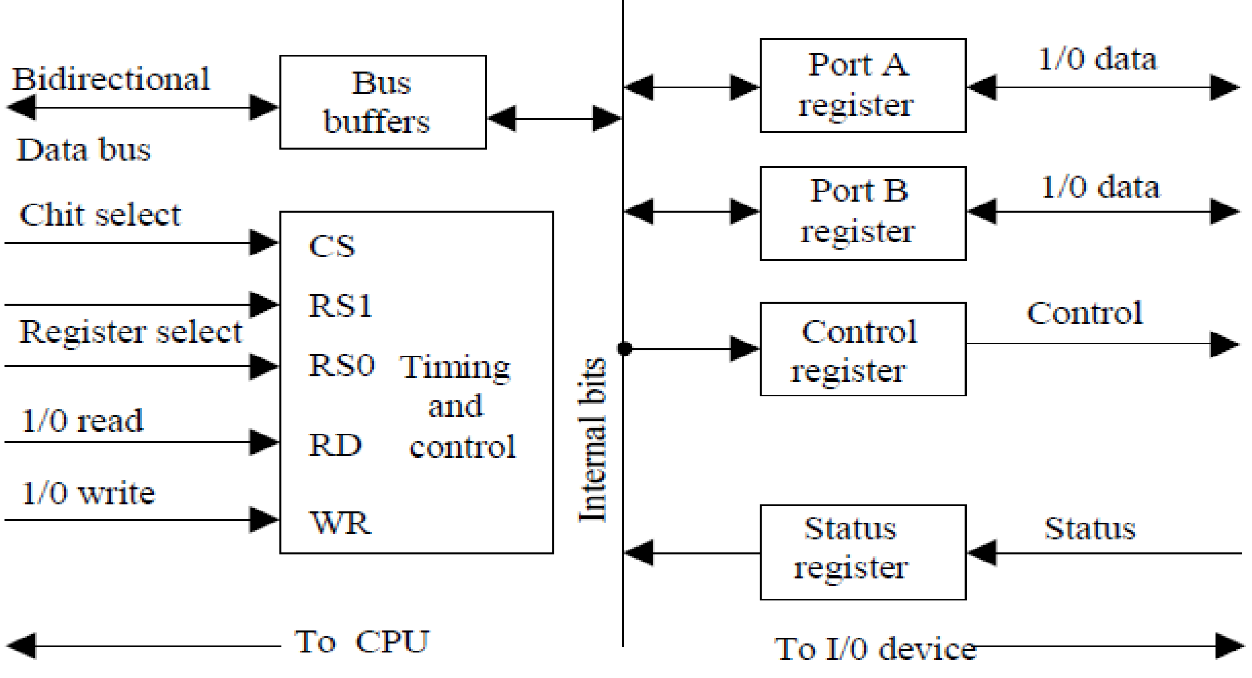
- In a **memory-mapped I/O** organization there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words.
- Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.
- Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers.

Input-output Interface

Example Of I/O Interface

- An example of an I/O interface unit is shown in block diagram form in Fig. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.
- The I/O data to and from the device can be transferred into either port A or Port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output.
- If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines.
- Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular register with which the CPU communicates.

Input-output Interface



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Figure: Example of I/O interface unit.

Asynchronous Data Transfer

- The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator.
- Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse.
- Two units, such as a CPU and an I/O interface, are designed independently of each other.
- Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted.
- One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.

Asynchronous Data Transfer

Strobe Control

- The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure (a) shows a source-initiated transfer.
- The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

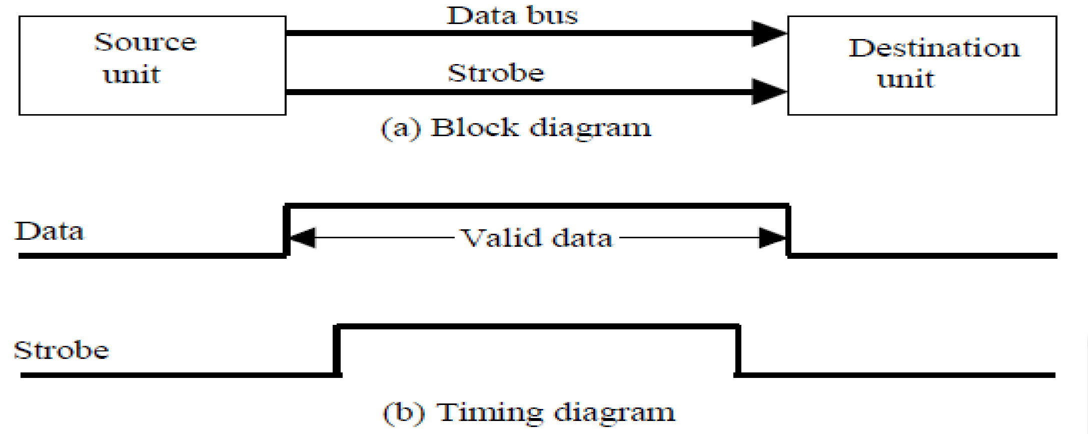


Figure: Source-initiated strobe for data transfer.

Asynchronous Data Transfer

- As shown in the timing diagram of Fig.(b), the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse.
- The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers.
- The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valued data. New valid data will be available only after the strobe is enabled again.

Asynchronous Data Transfer

- Below Figure shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it.
- The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus. Similarly, the strobe of fig.could be a memory-read control signal from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is the source, to place a selected word into the data bus.

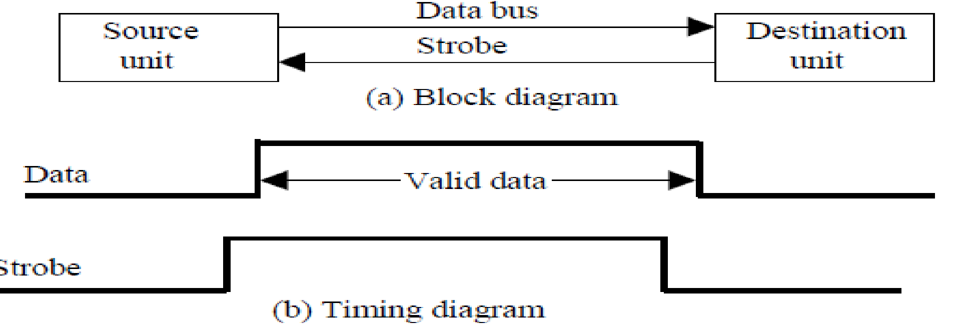


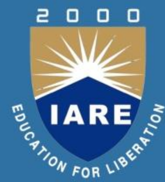
Figure: Destination-initiated strobe for data transfer.

Asynchronous Data Transfer

Handshaking

- The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus.
- Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus,. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer.
- Figure shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time.

Asynchronous Data Transfer



Source unit | Data bus → | Data valid → | ← Data accepted | Destination unit

(a) Block diagram

Data bus — Valid data
Data valid
Data accepted

(b) Timing diagram

Source unit | Destination unit

Place data on bus. Enable *data valid*.
Accept data from bus. Enable *data accepted*.
Disable *data valid*. Invalidate data on bus.
Disable *data accepted*. Ready to accept data (initial state).

(c) Sequence of events

Figure: source initiated transfer using handshaking.

Asynchronous Data Transfer

- The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus.
- The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal.

Asynchronous Data Transfer

Source unit | Data bus → | Data valid → | ← Ready for data | Destination unit

(a) Block diagram

Ready for data

Data valid

Data bus ← Valid data →

(b) Timing diagram

Source unit

Destination unit

Ready to accept data. Enable *ready for data*.

Place data on bus. Enable *data valid*.

Accept data from bus. Disable *ready for data*.

Disable *data valid*. Invalidate data on bus (initial state).

(c) Sequence of events

Figure: Destination initiated transfer using handshaking.

Asynchronous Data Transfer

- The destination-initiated transfer using handshaking lines is shown in Fig. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.
- The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking procedure follows the same pattern as in the source-initiated case.
- Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted. In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.

Modes Of Transfer

- 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.

Modes Of Transfer

- 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.

Modes Of Transfer

Example Of Programmed I/O

- An example of data transfer from an I/O device through an interface into the CPU is shown in 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 a 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.
- 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 from the 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.

Modes Of Transfer

- 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. 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 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.

Modes Of Transfer

Figure: Data transfer form I/O device to CPU

CPU
Data bus
Address bus
1/0 read
1/0 write
Interface
Data register
Status register
F
1/0 bus
Data valid
Data accepted
1/0 device
F = Flag bit

Read data register
Check flag bit
Flag
= 0
= 1
Read status register
Transfer data to memory
Operation complete?
no
yes
Continue with program

Figure: Flowchart for CPU program to input data.

Modes Of Transfer

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.

Modes Of 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.

Priority Interrupt

Daisy-chaining Priority

- The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain.
- This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt.
- If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

Priority Interrupt

- A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower priority device that the acknowledge signal has been blocked.
- A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device

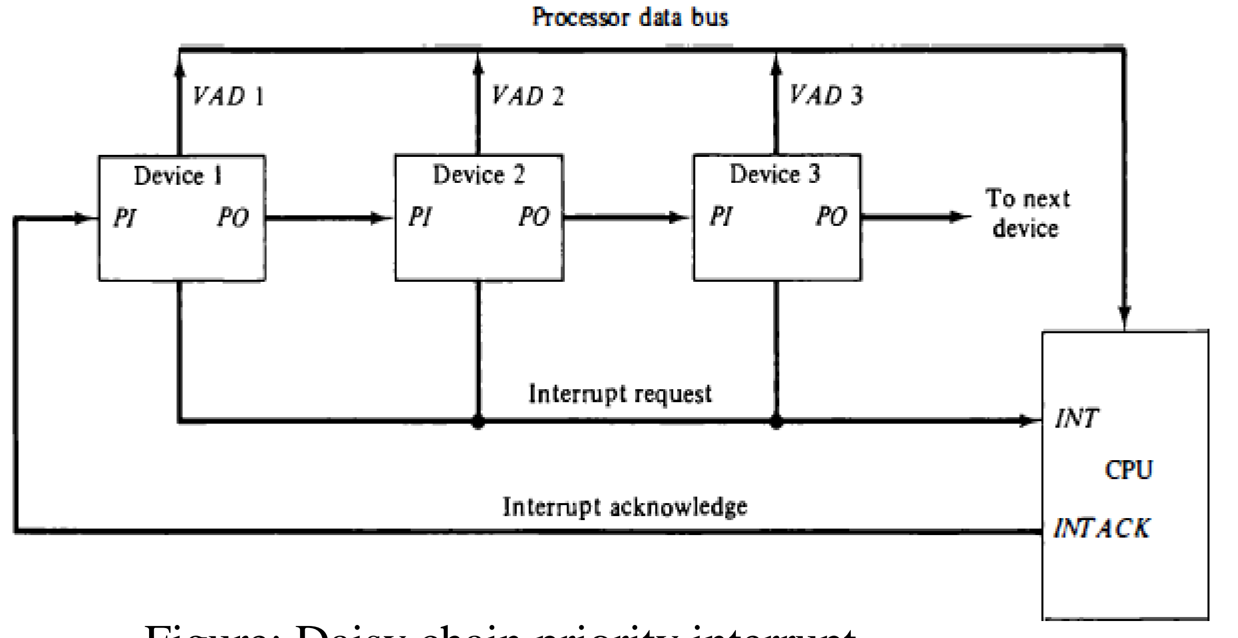


Figure: Daisy chain priority interrupt.

Priority Interrupt

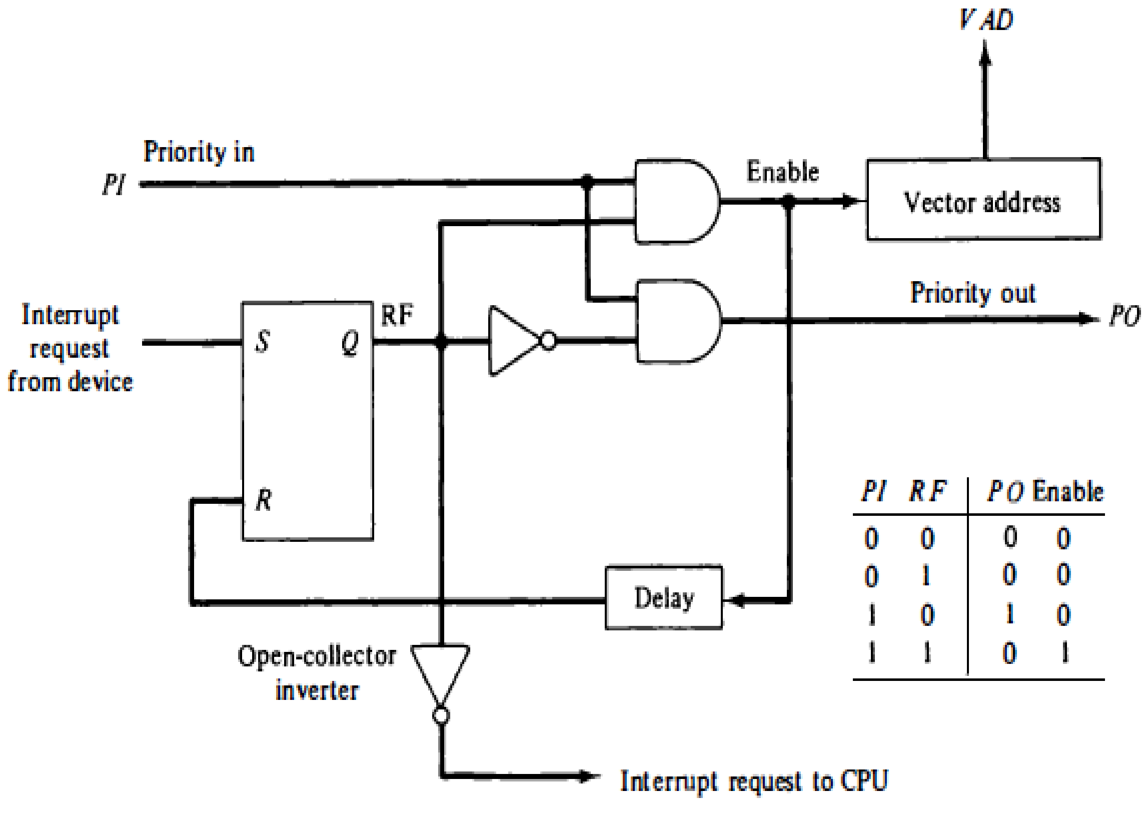
- by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus.
- The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Priority Interrupt

Parallel Priority Interrupt

- The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable.
- The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF.
- If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

Priority Interrupt



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

- lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

Priority Interrupt

priority logic

- The priority logic for a system of four interrupt sources is shown in Fig. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions.
- The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register.
- By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder.
- In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

Priority Interrupt

Interrupt register
Disk
Printer
Reader
Keyboard
0
1
2
3
I_0
I_1
I_2
I_3
Priority encoder
VAD to CPU
y
x
0
0
0
0
0
0
Enable
IEN
IST
Interrupt to CPU
INTACK from CPU
0
1
2
3
Mask register

Figure: Priority interrupt hardware.

Priority Interrupt

- Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.
- The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

Direct Memory Access (DMA)

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 shows two control signals in the CPU that facilitate the DMA transfer.

Bus request → BR
Bus grant ← BG
CPU
DBUS ↔ Data bus
ABUS → Address bus
RD → Read
WR → Write
High-impedance (disable) when BG is enabled

Figure : CPU bus signals for DMA transfer.

Direct Memory Access (DMA)

- 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 dies 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.

DMA Controller

- The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address register are used for direct communication with the memory.
- The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA. Figure shows the block diagram of a typical DMA controller.
- The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional.
- When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.

DMA Controller

- When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. the DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.
- The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.
- The word count register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.
- The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

DMA Controller

- The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

- The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

- 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.
- 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 and the memory. 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.

DMA Transfer

- When BG = 1, the RD and WR and output lines from the DMA controller to the random-access 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.

DMA Transfer

Interrupt
BG
BR
CPU
RD WR Address Data

Random-access memory (RAM)
RD WR Address Data

Read control
Write control
Data bus
Address bus

Address select

RD WR Address Data
DS
RS
BR
BG
Interrupt
Direct memory access (DMA) controller

DMA acknowledge
DMA request

I/O Peripheral device

Figure: DMA transfer in a computer system.

DMA Transfer

- For each word that is transferred, the DMA increments its address register 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.

MODULE-V

PIPELINE: PARALLEL PROCESSING

Course Outcomes

CO 1	Understand the Pipelining and Parallel processing units.
CO 2	Classify the pipelining, arithmetic pipeline and instruction pipeline.
CO 3	Describe the Characteristics of multiprocessors and inter connection structures.
CO 4	Classify the different inter processor arbitration, inter processor communications and synchronization.

Contents

Pipeline:

- Parallel processing
- Pipelining-arithmetic pipeline
- Instruction pipeline

Multiprocessors:

- Characteristics of multiprocessors
- Inter connection structures
- Inter processor arbitration
- Inter processor communication and synchronization.

Pipelining

- Pipelining is a technique of decomposing a sequential process into sub operations,
- with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
- **Throughput**: The amount of processing that can be accomplished during a given interval of time .

Parallel Processing

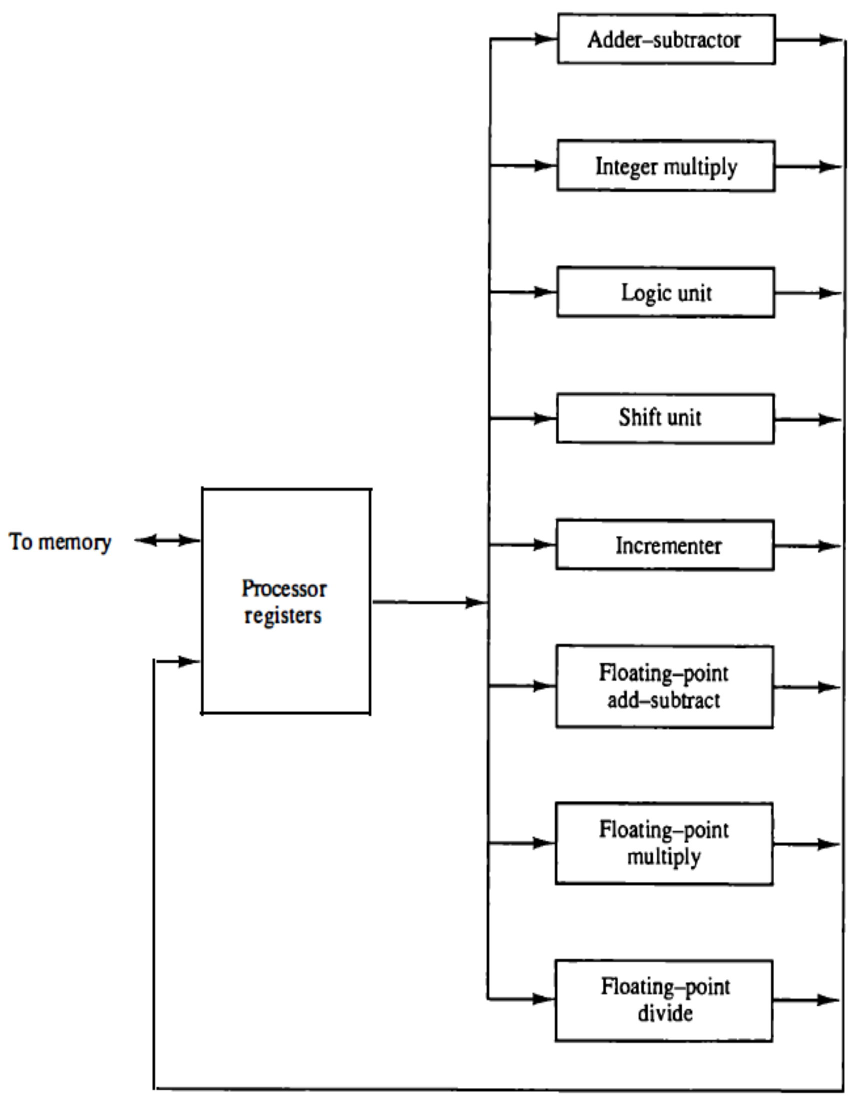


Fig: Processor with Multiple Functional Units

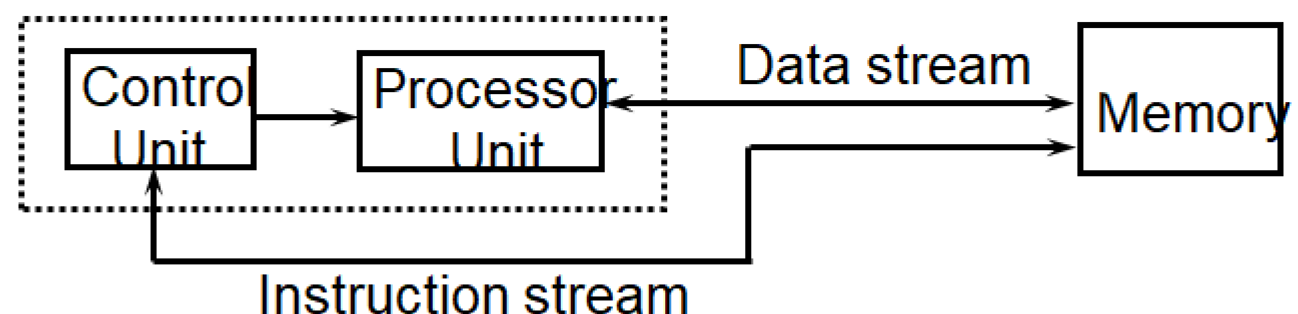
Parallel Processing

- M.J.Flynn Classify the parallel processing based on the number of instructions and data items that are manipulated simultaneously .
- Instruction Stream
  - Sequence of Instructions read from memory .
- Data Stream
  - Operations performed on the data in the processor .
- Flynn's Classification divides computers into four major groups:

		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	SISD	SIMD
	Multiple	MISD	MIMD

Parallel Processing

SISD Computer Systems



- Characteristics
  - Standard von Neumann machine
  - Instructions and data are stored in memory
  - One operation at a time
  - Parallel processing is achieved by means of multiple functional units or by pipeline.

- Limitations
  - Von Neumann bottleneck
  - Maximum speed of the system is limited by the *Memory Bandwidth*
  - Limitation on *Memory Bandwidth*
  - Memory is shared by CPU and I/O

MISD Computer System

M → CU → P
M → CU → P
M → CU → P

Memory

Data stream

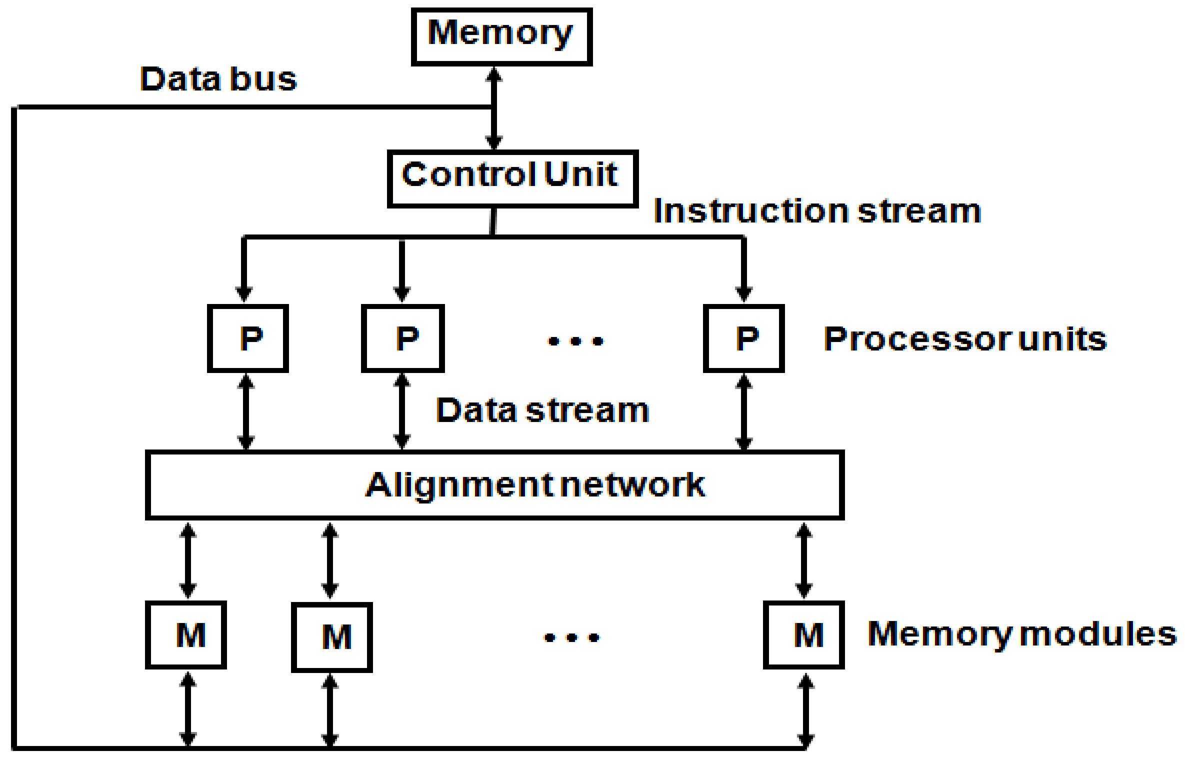
Instruction stream

Characteristics

- There is no computer at present that can be classified as MISD .

Parallel Processing

SIMD Computer System

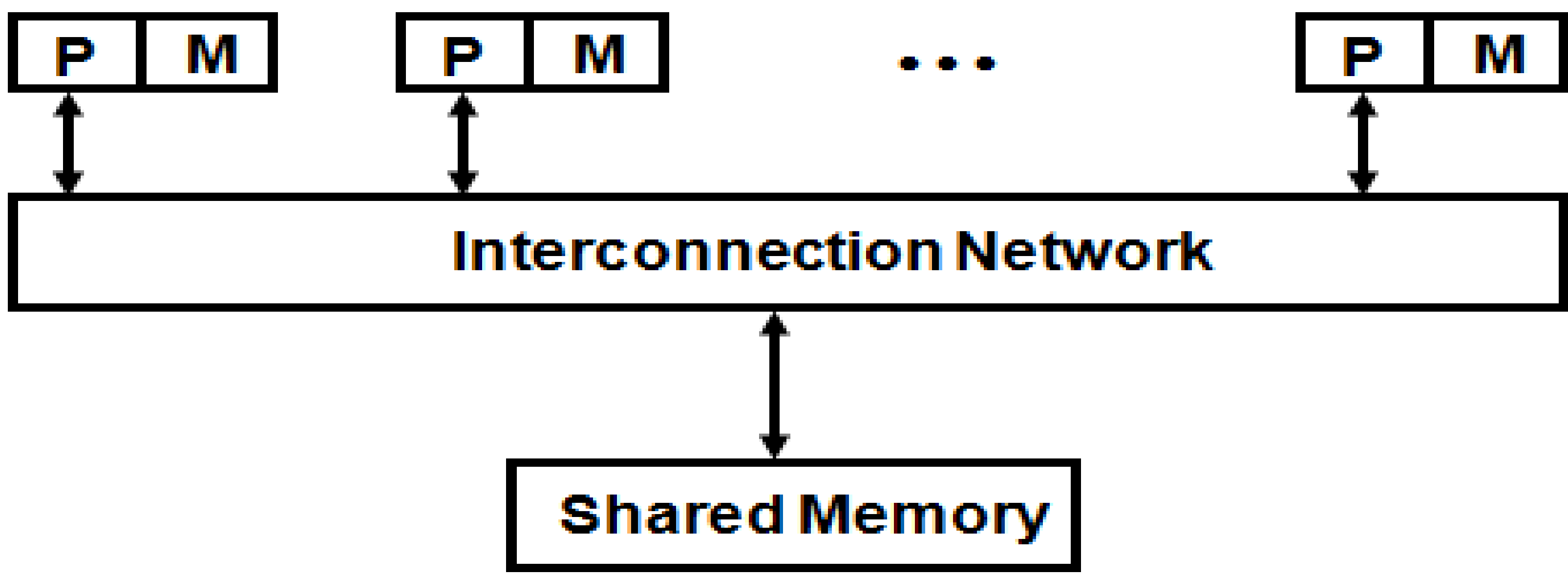


Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time

Parallel Processing

MIMD Computer System



Characteristics

- Multiple processing units
- Execution of multiple instructions on multiple data.
- Multiprocessors and multi computers are MIMD computers.

Parallel Processing

- The parallel processing Is achieved by
  1)Pipeline Processing
  2) Vector Processing
  3) Array processors

Pipelining :

- A technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

Example

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7

$R1 \leftarrow A_i, \ R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2, \ R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

Parallel Processing

Pipelining

Segment 1

Segment 2

Segment 3

Memory A_i B_i C_i

R1 R2

Multiplier

R3 R4

Adder

R5

Fig: Pipe Line Processing

Parallel Processing

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

Table: Content of register in pipeline

General Pipeline

Clock

Input → S_1 → R_1 → S_2 → R_2 → S_3 → R_3 → S_4 → R_4 →

Fig : Four-Segment Pipeline

General Pipeline

	1	2	3	4	5	6	7	8	9
Segment 1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

→ Clock cycles

Fig: Space-Time Diagram for pipeline

Parallel Processing

Pipeline Speedup

n: Number of tasks to be performed

K: number of segments

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

t_1: Time required to complete the n tasks

$t_1 = n * t_n$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

t_k: Time required to complete the n tasks

$t_k = (k + n - 1) * t_p$

Speedup

S_k: Speedup

$S_k = n*t_n / (k + n - 1)*t_p$

Parallel Processing

Pipeline Speedup

Example

- 4-stage pipeline
- sub opertion in each stage; $t_p = 20nS$
- 100 tasks to be executed
- 1 task in non-pipelined system; 20*4 = 80nS

Pipelined System

$$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$$

Non-Pipelined System

$$n*k*t_p = 100 * 80 = 8000nS$$

Speedup

$$S_k = 8000 / 2060 = 3.88$$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Parallel Processing

General Pipeline

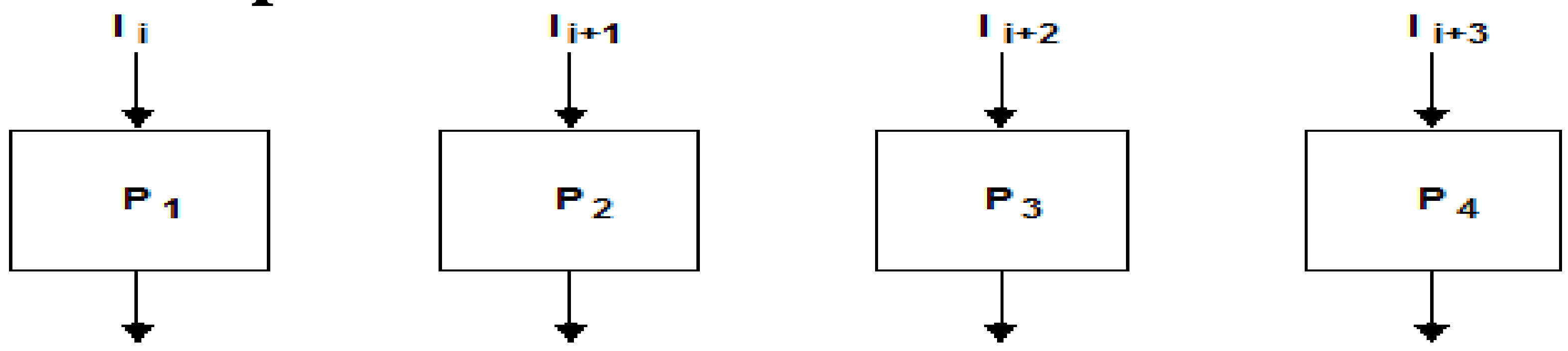


Fig: Multiple Functional Units

There are two are as of computer design where the pipeline organization is use full.

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

- The Arithmetic pipeline divides an arithmetic operation into sub operations for execution in the pipeline segments.
- The inputs for the floating point adder pipeline :

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

- Here A and B are the Fractions that represents the mantissa and a and b are the exponents.
- The floating point addition and subtraction divided into four segments:

1.Compare the exponents

2.Align the mantissa

3.Add or subtract the mantissa

4.Normalize the result

Arithmetic Pipeline

- The following numerical example may clarify the sub operations performed in each segment. For simplicity, we use decimal numbers, although Fig. refers to binary numbers. Consider the two normalized floating-point numbers:

  X = 0. 9504 X 10'

  Y = 0.8200 X 1<J1

- The two exponents are subtracted in the first segment to obtain 3 - 2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

  X = 0.9504 X 103

  Y = 0. 0820 X 103

- This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

  Z = 1 . 0324 X 10‘

- The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

  Z = 0.10324 X 10'

Arithmetic Pipeline

Exponents
a b
R
Segment 1: Compare exponents by subtraction
Difference
R
Segment 2: Choose exponent
Segment 3:
R
Segment 4: Adjust exponent
R

Mantissas
A B
R
Align mantissa
R
Add or subtract mantissas
R
Normalize result
R

Fig : Pipeline for Floating point addition and Subtraction

Arithmetic Pipeline

- The comparator, shifter, adder-sub tractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are t, = 60 ns, t2 = 70 ns, t3 = 100 ns, t4 = 80 ns, and the interface registers have a delay of t, = 10 ns.
- The clock cycle is chosen to be t, = t3 + t, = 110 ns. An equivalent non pipeline floating point adder-sub tractor will have a delay time t, = t, + t2 + t, + t4 + t, = 320 ns. In this case the pipelined adder has a speedup of 32011 10 = 2. 9 over the Non pipelined adder

Instruction Pipeline

- Pipeline processing can occur not only in the data stream but in the instruction stream as well.
- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.
- This causes the instruction fetch and execute phases to overlap and perform simultaneous operations .
- One disadvantage with this scheme is that an instruction may cause a branch out of sequence. In this case the pipeline must be empted and all instructions read from memory after branch.
- A computer with an instruction fetch unit and an instruction execution unit designed to provide a two segment pipeline.

Instruction Pipeline

- In general any computer system needs six steps to process any instruction .

  1 . Fetch an instruction from memory

  2 . Decode the instruction

  3 . Calculate the effective address of the operand

  4 . Fetch the operands from memory

  5 . Execute the operation

  6 . Store the result in the proper place

- Some instructions skip some phases

  - Effective address calculation can be done in the part of the decoding phase .

  - Storage of the operation result into a register is done automatically in the execution phase .

Instruction Pipeline

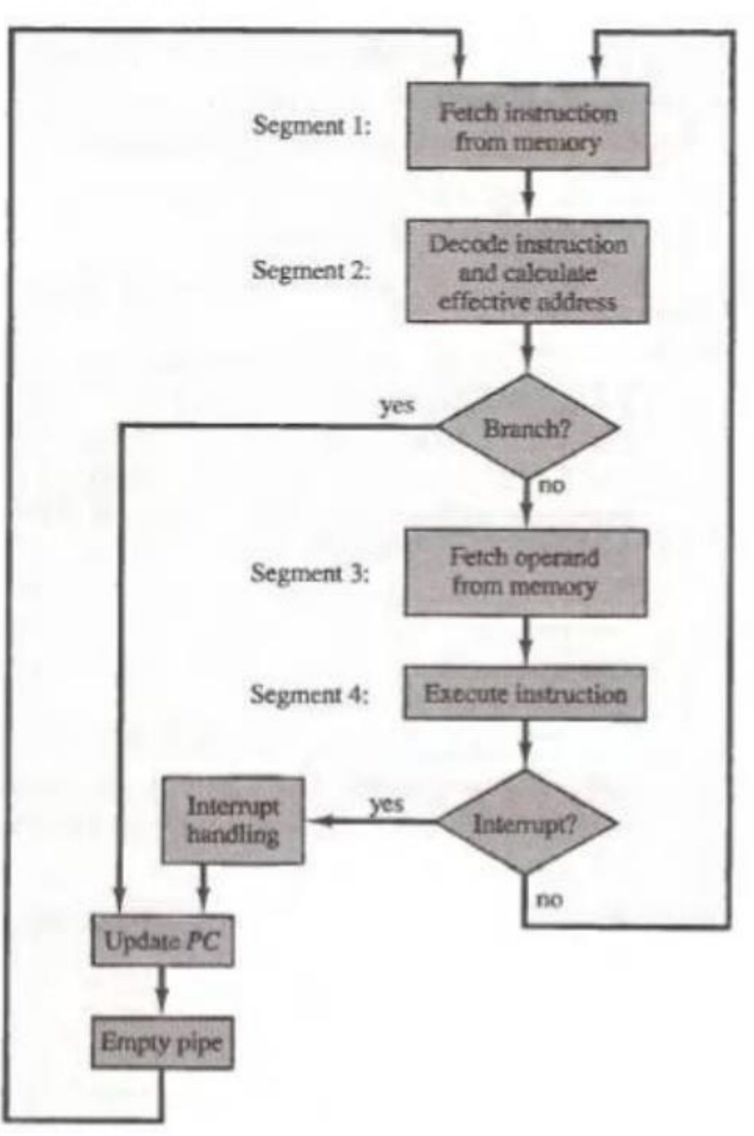


Fig: Four-segment CPU pipeline

Instruction Pipeline

- Reduces the instruction pipeline into four segments. Figure shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.
- The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.
- Thus up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.
- Figure shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

Instruction Pipeline

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

- It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.
- Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Instruction Pipeline

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure: Timing of instruction pipeline.

pipeline conflicts:

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. *Resource conflicts* caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

2. *Data dependency* conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

3. *Branch difficulties* arise from branch and other instructions that change the value of PC .

Characteristics of Multiprocessors

Parallel Computing

- Simultaneous use of multiple processors, all components of a single architecture, to solve a task.
- Typically processors identical, single user .

Distributed Computing

- Use of a network of processors, each capable of being viewed as a computer in its own right, to solve a problem.
- Processors may be heterogeneous, multiuser, usually individual task is assigned to a single processors .

Concurrent Computing

- All of the above

Characteristics of Multiprocessors

Supercomputing

- Use of fastest, biggest machines to solve big, computationally intensive problems.
- Historically machines were vector computers, but parallel/vector or parallel becoming the norm.

Pipelining

- Breaking a task into steps performed by different units, and multiple inputs stream through the units, with next input starting in a unit when previous input done with the unit but not necessarily done with the task.

Vector Computing

- Use of vector processors, where operation such as multiply broken into several steps, and is applied to a stream of operands ("vectors"). Most common special case of pipelining.

Characteristics of Multiprocessors

Multiprocessor computer

- Execute a number of different application tasks in parallel
- Execute subtasks of a single large task in parallel
- All processors have access to all of the memory – shared-memory multiprocessor
- Cost – processors, memory units, complex interconnection networks

- Reliable, Single OS

Multicomputers

- Each computer only have access to its own memory
- Exchange message via a communication network – message-passing multicomputers

- Reliable ,Different OS

Characteristics of Multiprocessors

Multiprocessors and Multicomputers

- To achieve benefit of multiprocessing the computations can be processed in two ways
  1) multiple independent jobs can be made.
  2) A single job can be partitioned into multiple parallel tasks.
- Multiprocessors are classified into two types based on their memory organization.

1) Tightly Coupled System

- Tasks and/or processors communicate in a highly synchronized fashion
- Communicates through a common shared memory
- Each processor may have cache memory.
- Shared memory system
- Efficient when higher degree of interaction between tasks.

Characteristics of Multiprocessors

Multiprocessors and Multicomputers

2) Loosely Coupled System

- Tasks or processors do not communicate in a synchronized fashion
- Each processor has its own private local memory.
- Information is shared among processor through message passing scheme.
- A packet consists of an address, the data content and some error detection code.
- Distributed memory system
- Efficient when the interaction between the tasks are minimal.

Characteristics of Multiprocessors

MEMORY

– Shared (Global) Memory

  - A Global Memory Space accessible by all processors
  - Processors may also have some local memory

– Distributed (Local, Message-Passing) Memory

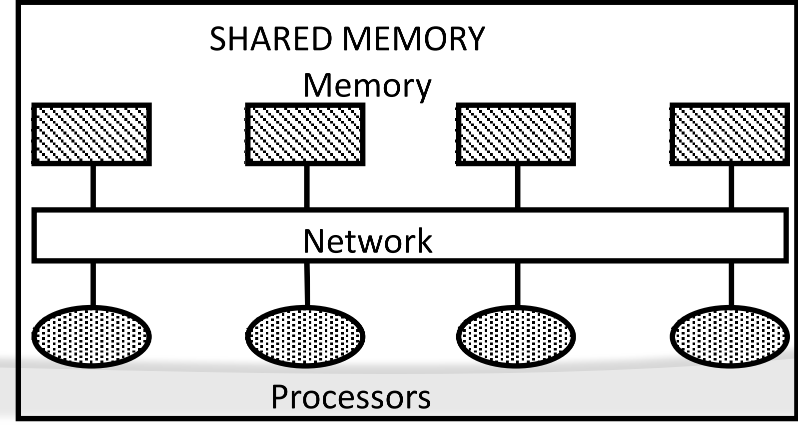
  - All memory units are associated with processors
  - To retrieve information from another processor's memory a message must be sent there

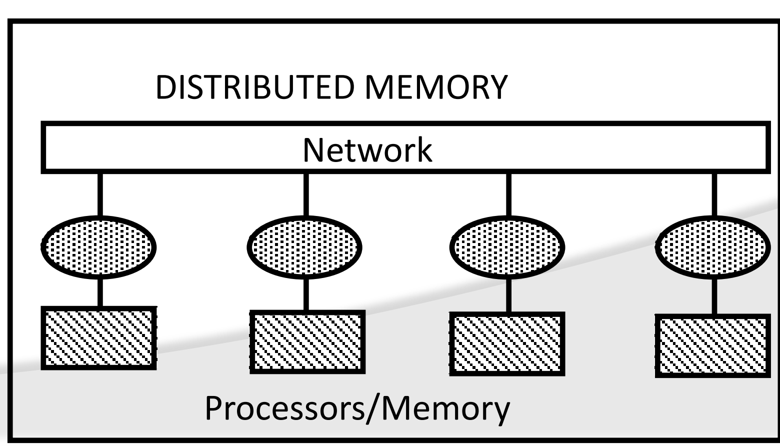
– Uniform Memory

  - All processors take the same time to reach all memory locations

– Nonuniform (NUMA) Memory

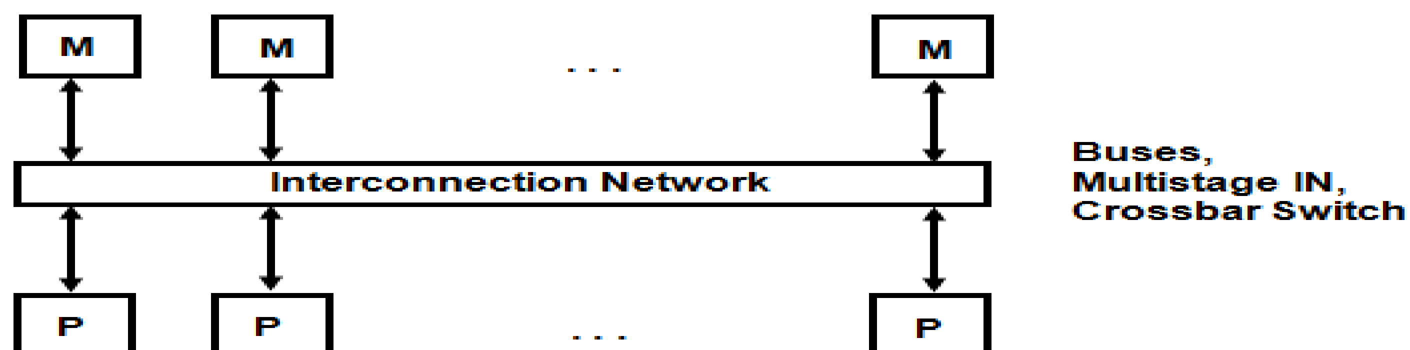
  - Memory access is not uniform





Characteristics of Multiprocessors

Shared Memory Multiprocessors



Characteristics

- All processors have equally direct access to one
- large memory address space

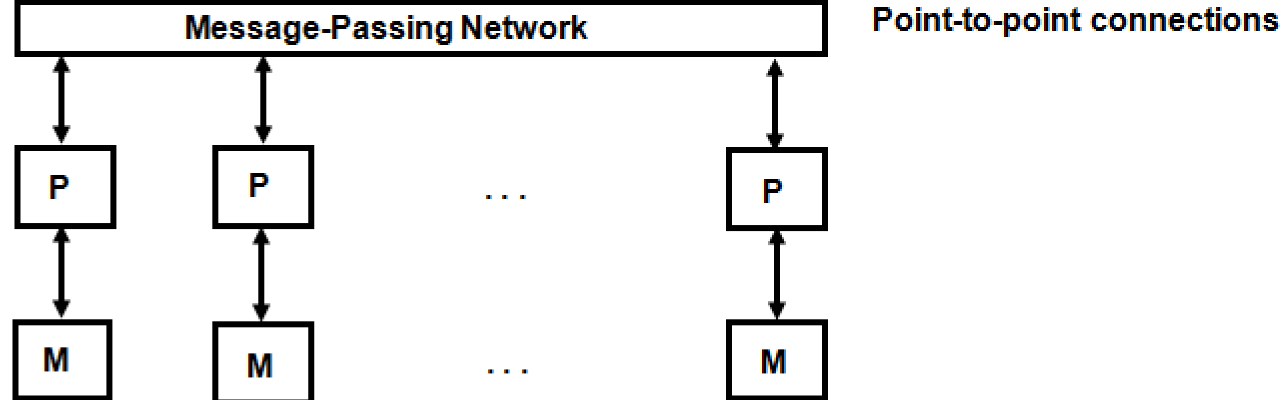
Example systems

- Bus and cache-based systems: Sequent Balance, Encore Multimax
- Multistage IN-based systems: Ultracomputer, Butterfly, RP3, HEP
- Crossbar switch-based systems: C.mmp, Alliant FX/8

Limitations

- Memory access latency; Hot spot problem

Message-Passing Multiprocessors



Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

Limitations

- Communication overhead; Hard to programming

Interconnection Structures

1) Time-Shared Common Bus
2) Multiport Memory
3) Crossbar Switch
4) Multistage Switching Network
5) Hypercube System

Time-Shared Common Bus

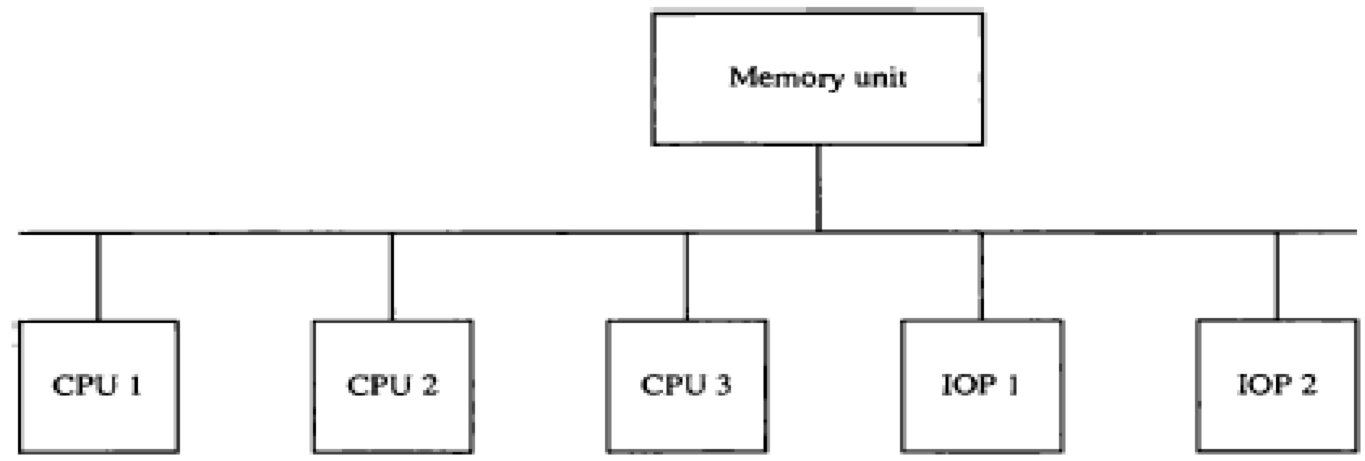


Figure: **Time-shared** common bus organization.

- A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig. Only one processor can communicate with the memory or another processor at any given time.

Time-Shared Common Bus

- Transfer operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer.
- A command is issued to inform the destination unit what operation is to be performed . The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated.
- A single common-bus system is restricted to one transfer at a time

Interconnection Structures

Local Bus

Common Shared Memory

System Bus Controller

CPU

IOP

Local Memory

SYSTEM BUS

System Bus Controller

CPU

IOP

Local Memory

System Bus Controller

CPU

Local Memory

Local Bus

Local Bus

Fig : system bus structure for multiprocessors

Interconnection Structures

- Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus . The IO devices connected to the local IOP, as well as the local memory, are available to the local processor.
- The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus, the VO devices attached to it may be made available to all processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time.
- The other processors are kept busy communicating with their local memory and IO devices. Part of the local memory may be designed as a cache memory attached to the CPU in this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

Multiport Memory

- A multiport memory system employs separate buses between each memory module and each CPU.
- for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module. A processor bus consists of the address, data, and control lines required to communicate with memory. The memory module is said to have four ports and each port accommodates one of the buses.
- Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.
- The advantage of the multi port memory organization is the higher rate that can be achieved because of the multiple paths between processors and memory. The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors.

Interconnection Structures

Memory Modules

MM 1 | MM 2 | MM 3 | MM 4

CPU 1
CPU 2
CPU 3
CPU 4

Fig: Multiport Memory

Crossbar Switch

- The cross bar switch consists of a number of cross points that are placed at intersection between processor bus and memory module paths.
- The small square in each cross point is a switch that determines the path from a processor to memory module.
- Each switch contains control logic to set up the transfer path between a processor and memory.
- It determines the address and it resolves the multiple requests for access to the same memory module on a predetermined priority basis.
- A cross bar switch organization support simultaneous transfers from all memory modules.

Interconnection Structures

Memory modules

MM1 MM2 MM3 MM4

CPU1

CPU2

CPU3

CPU4

Fig : Crossbar Switch

Interconnection Structures

Memory Module

data

address

R/W

memory enable

Multiplexers and arbitration logic

data,address, and control from CPU 1

data,address, and control from CPU 2

data,address, and control from CPU 3

data,address, and control from CPU 4

Fig: Block Diagram of Crossbar Switch

Multistage Switching Network

- The basic component of a multistage network is a two input , two output inter change switch.
- The 2 X 2 Switch has two inputs and A and B and Two outputs 0 and 1.

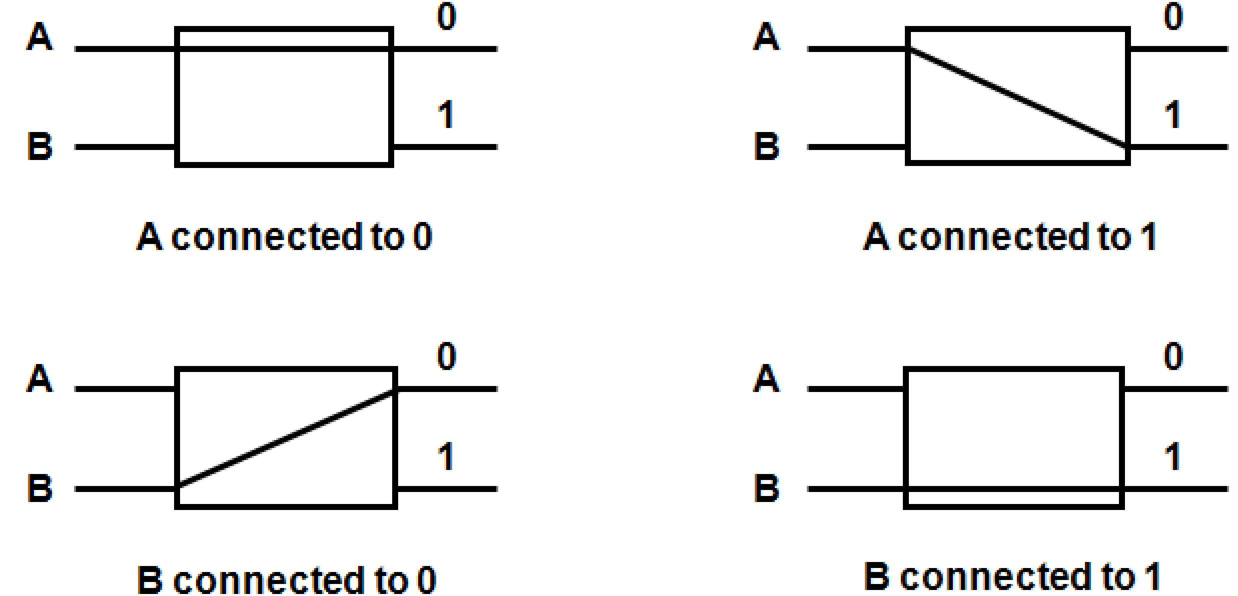


Fig : operation of 2 X 2 Switch

Interconnection Structures

P1
P2
0
1
0
1
0
1
0 000
1 001
0 010
1 011
0 100
1 101
0 110
1 111

connectors.

Fig : Binary tree with 2 X 2 switch

Interconnection Structures

0
1
2
3
4
5
6
7

000
001
010
011
100
101
110
111

Fig : 8 X 8 Omega Switching Network

Hypercube Interconnection

- The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N=2^n$ processors interconnected in an n dimensional binary cube.
- each processor forms a node of the cube.
- In each node it contains processor ,memory and I/O Interface.
- Each processor has direct communication paths to n other neighbor processors.
- There are 2^n distinct n-bit binary address that can be assigned to the processors.
- Each processor address differs from that of each of its n neighbors by exactly one bit position.
- The routing procedure developed by using XOR operation.

Interconnection Structures

0
1

01 11
00 10

011 111
010 110
001 101
000 100

One-cube   Two-cube   Three-cube

Fig: Hypercube Structure for n=1,2,3

- There are two arbitrations
  1. Serial (daisy-chain) arbitration.
  2. Parallel Arbitration

Serial (daisy-chain) arbitration.

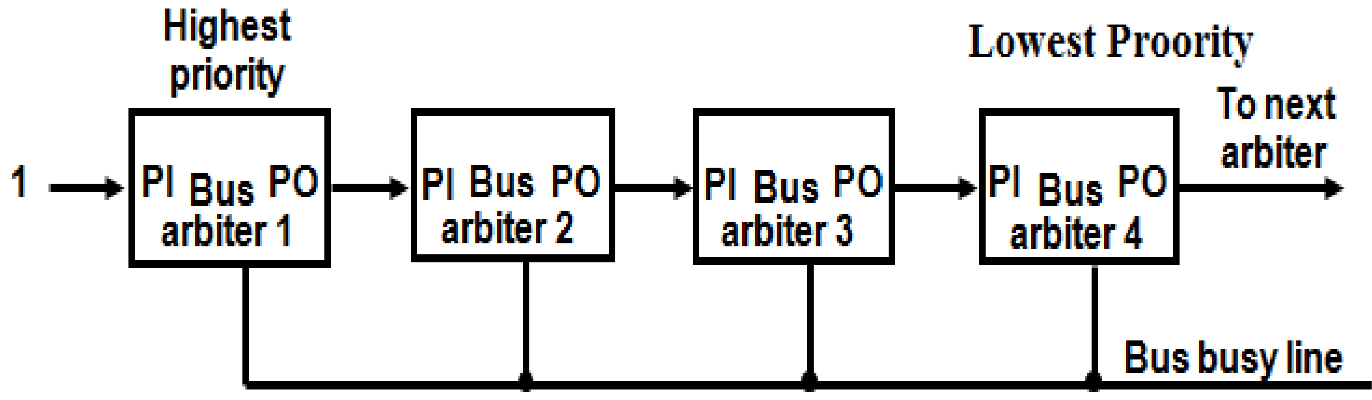


Fig : Serial Arbitration

Inter Processor Arbitration

- The processors connected to the system bus are assigned priority according to their position along the priority control line.
- The device closest to the priority line is assigned the highest priority.
- When multiple devices concurrently request the use of the bus , the device with the highest priority is granted access to it.
- In the above figure shows the daisy chain connection of four arbiters. Each processor contains its own arbiter logic with priority in and priority out .
- The Priority out of each processor (PO) is connected to the priority in (PI) of the next level priority arbiter.
- The PI of the highest-priority unit is maintained at a logic 1 value.

Inter Processor Arbitration

- The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus.
- If the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1 , it sets its PO output to 0.
- Lower-priority arbiters receive a 0 in PI and generates a 0 in PO.
- The processor whose arbiter has a PI=1 and PO =0 is the one that is given the control of the system bus.
- A processor may be in the middle of a bus operation when a higher priority processor request the bus. The lower-priority processor must complete its bus operation before it release control of the bus.

Inter Processor Arbitration

Parallel Arbitration Logic

Fig: Parallel Arbitration

Inter Processor Arbitration

- The parallel bus arbitration technique uses an external priority encoder and a decoder.
- Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line.
- Each arbiter enables the request line when its processor is requesting access to the system bus.
- The processor takes control of the bus if its acknowledge input line is enabled.

Inter processor communication and synchronization

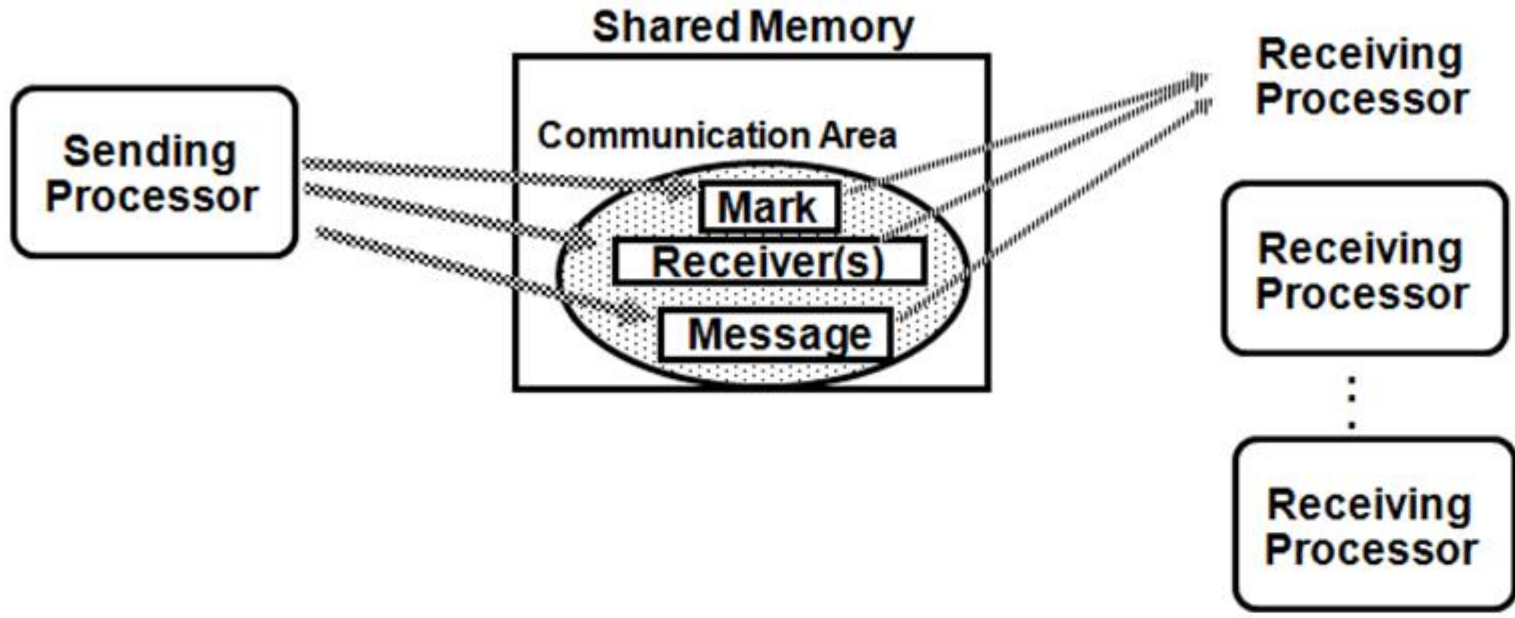


Fig: Shared Memory Communication

Inter processor communication and synchronization

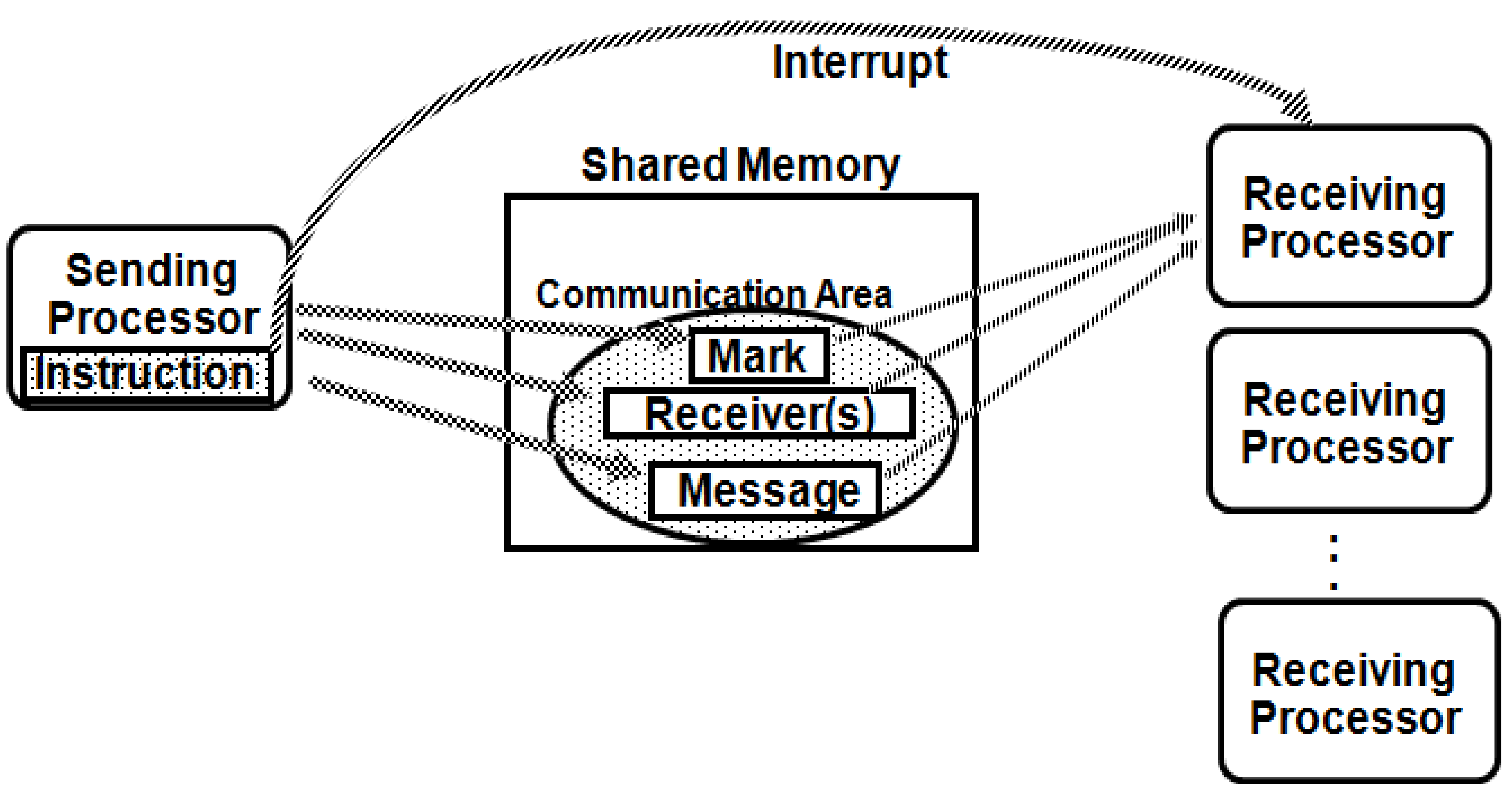


Fig: Shared Memory Communication with Interrupt

Inter processor communication and synchronization

Interprocessor Synchronization

In multiprocessor system communication refers to the exchange of data between different processes.

Synchronization is the process of where the data used to communicate between processor is control information.

Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.

Mutual Exclusion with a Semaphore

Mutual Exclusion

One processor to exclude or lock out access to shared resource by other processors when it is in a *Critical Section* .

Critical Section

is a program sequence that once begun, must Complete execution before another processor accesses the same shared resource.

Inter processor communication and synchronization

IARE

Semaphore

A binary variable

- 1: A processor is executing a critical section, that not available to other processors .
- 0: Available to any requesting processor

Software controlled Flag that is stored in memory that all processors can be access .

Testing and Setting the Semaphore

Avoid two or more processors test or set the same semaphore

May cause two or more processors enter the same critical section at the same time .

Must be implemented with an indivisible operation

```
R <- M[SEM]        / Test semaphore /
M[SEM] <- 1        / Set semaphore /
```

Inter processor communication and synchronization

- These are being done while *lock*ed, so that other processors cannot test and set while current processor is being executing these instructions.
- If R=1, another processor is executing the critical section, the processor executed this instruction does not access the shared memory .
- If R=0, available for access, set the semaphore to 1 and access The last instruction in the program must clear the semaphore .

Inter processor communication and synchronization

IARE
2000
EDUCATION FOR LIBERATION

- Communication of control information between processors
  to enforce the correct sequence of processes
- To ensure mutually exclusive access to shared writable data
- Hardware Implementation
- Mutual Exclusion with a Semaphore

Inter processor communication and synchronization

IARE
2000
EDUCATION FOR LIBERATION

- Mutual Exclusion
- One processor to exclude or lock out access to shared resource by other processors when it is in a Critical Section
- Critical Section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource

Inter processor communication and synchronization

IARE EDUCATION FOR LIBERATION 2000

Semaphore

•binary variable

1: A processor is executing a critical section, that not available to other processors

0: Available to any requesting processor software controlled Flag that is stored in memory that all processors can be access

Inter processor communication

- Testing and Setting the Semaphore
- Avoid two or more processors test or set the same **semaphore**
- May cause two or more processors enter the
- same critical section at the same time
- Must be implemented with an indivisible operation

Cache coherence

- Shared data leads to another problem in a multiprocessor machine in the presence of multiple caches means that copies of shared data may reside in several caches.
- When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value.
- They must be informed of the change so that they can either update their copy to the new value or invalidate it.
- Cache coherence is defined as the situation in which all cached copies of shared data have the same value at all times.

1)Write Through:

1)Broad casting of the new value to all caches.

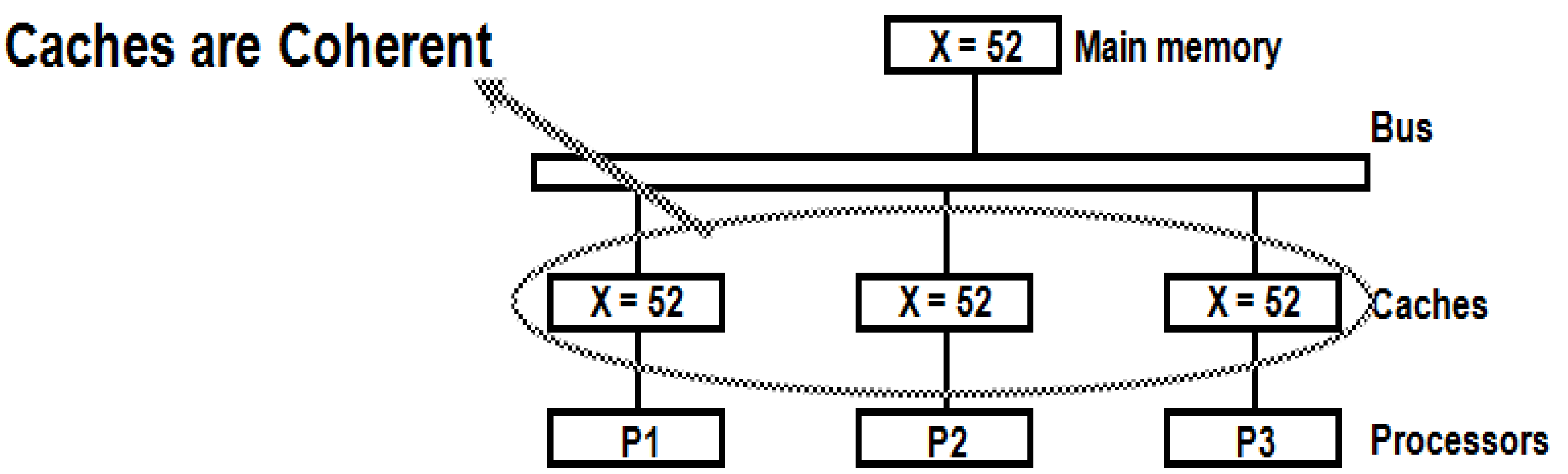
2)Invalidation of copies.

2)Write Back :

- Master slave mechanism

- Invalidate all other caches including memory.

Cache coherence



Cache Incoherency in
Write Through Policy

X = 120
Main memory
Bus
X = 120
X = 52
X = 52
Caches
P1
P2
P3
Processors

Cache coherence

Cache Incoherency in Write Back Policy

X = 52 Main memory

Bus

X = 120

X = 52

X = 52

Caches

P1

P2

P3

Processors