

# 2011 Computer Organization II one workshop practice

## UNIT-1

### INTRODUCTION TO COMPUTER ORGANIZATION

- Basic Computer Organization -- CPU Organization – Memory Subsystem Organization and Interfacing  
– I/O Subsystem Organization and Interfacing – A Simple Computer- Levels of Programming Languages, Assembly Language Instructions, Instruction Set Architecture Design, A simple Instruction Set Architecture.

#### 1.1 BASIC COMPUTER ORGANIZATION:

Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memory subsystem
- I/O subsystem.

The generic organization of these components is shown in the figure below.

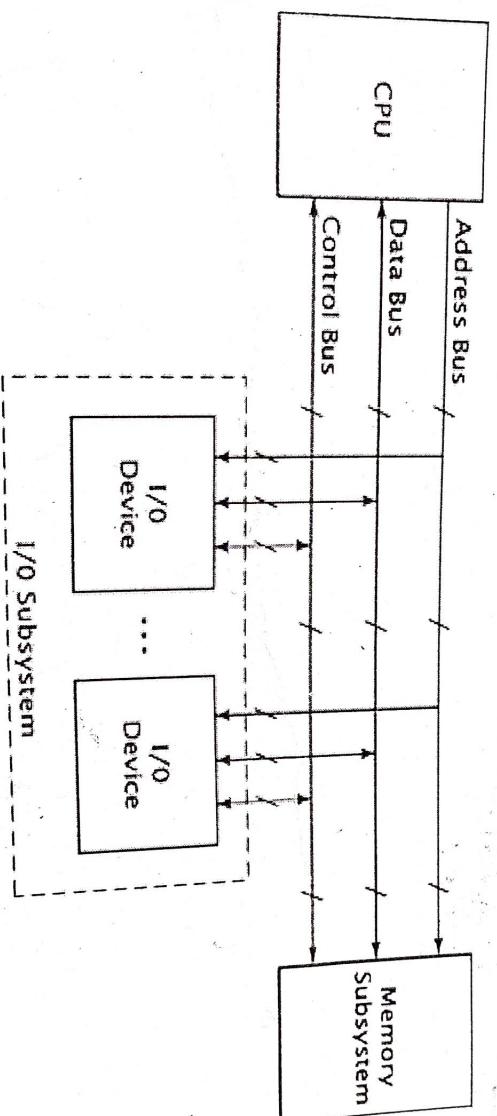


Fig 1.1 Generic computer Organization

#### 1.1.1 System bus:

Physically the bus a set of wires. The components of a computer are connected to the buses. To send information from one component to another the source component outputs data on to the bus. The destination component then inputs the data from bus.

The system has three buses

- Address bus
- Data bus
- Control bus

- The uppermost bus in this figure 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 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** 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.

### 1.1.2 Instruction cycles:

- The instruction cycle is the procedure a microprocessor goes through to process an instruction.
- First the processor **fetches** or reads the instruction from memory. Then it decodes the instruction determining which instruction it has fetched. Finally, it performs the operations necessary to execute the instruction.
- After fetching it **decodes** the instruction and controls the execution procedure. It performs some Operation internally, and supplies the address, data & control signals needed by memory & I/O devices to **execute** the instruction.
- 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.

Below figure shows the memory read and memory write operations.

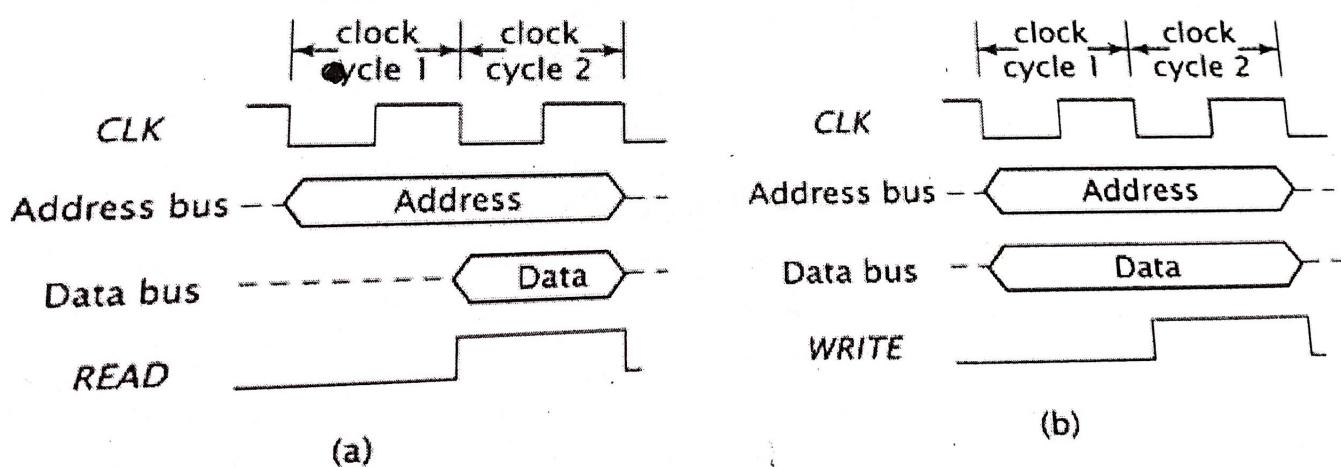


Fig 1.2: Timing diagram for memory read and memory write operations

- In the above figure the top symbol is CLK. This is the computer system clock. The processor uses the system clock to synchronize its operations.
- 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 deasserts the READ signal. Memory then removes the data from the data bus completing the memory read 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.
- I/O read and write operations are similar to the memory read and write operations. Basically the processor may use memory mapped I/O and isolated I/O.
- In memory mapped I/O it follows the same sequence of operations to input data as to read from or write data into memory.

In isolated I/O follow same process but have a second control signal to distinguish between I/O and memory accesses. For example in 8085 microprocessor has a control signal called IO/ . The processor set IO/ to 1 for I/O read and write operations and 0 for memory read and write operations.

## 1.2 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 major components are connected with the help of the **system bus**. **Data bus** is used to shuffle data between the various components in a computer system.

To differentiate memory locations and I/O devices the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device it places the corresponding address on the **address bus**. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on the data bus. Only the device whose address matches the value on the address bus responds.

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.

When both contain logic one the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero) the CPU is reading data from memory (that is the system is transferring data from memory to the CPU). If the write line is low the system transfers data from the CPU to memory.

The CPU controls the computer. It **fetches** instructions from memory, supply the address and control signals needed by the memory to access its data.

Internally, CPU has three sections as shown in the fig below

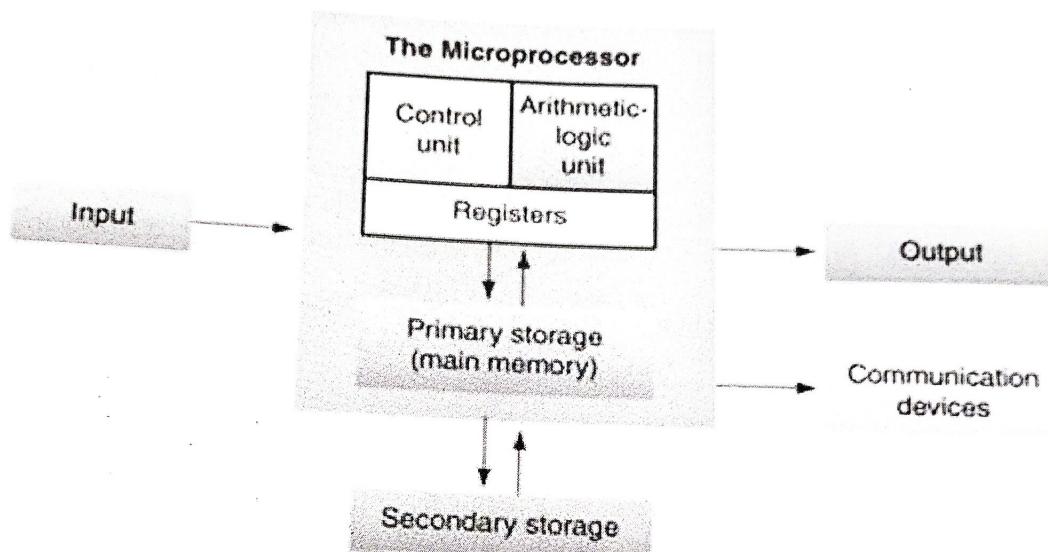


Fig 1.3: CPU Organization

- 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**".
- The CPU keeps the address of the next instruction to be fetched in this register. Before the CPU outputs the address on to the system bus, it retrieves the address from the program counter register.
- 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**".
- The arithmetic / logic unit (or) ALU performs most arithmetic and logic operations such as adding and ANDing values. It receives its operands form the register section of the CPU and stores its result

- back in the register section.
- Just as 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. This code generates the instruction codes & the values of some flag registers.
- The control unit also generates the signals for the system control bus such as READ, WRITE, IO/ M signals.

### **1.3 MEMORY SUBSYSTEM ORGANIZATION AND INTERFACING:**

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.

The size of the memory chip is denoted as the number of locations times the number of bits in each location. For example, a memory chip of size  $512 \times 8$  has 512 memory locations, each of which has eight bits. The address inputs of a memory chip choose one of its locations. A memory chip with  $2^n$  locations requires n address inputs.

- View the memory unit as a black box. Data transfer between the memory and the processor takes place through the use of two registers called MAR (Memory Address Register) and MDR (Memory data register).
- MAR is n-bits long and MDR is m-bits long, and data is transferred between the memory and the processor. This transfer takes place over the processor bus.

#### **Internal organization of the memory chips:**

- Memory is usually organized in the form of arrays, in which each cell is capable of storing one bit information.
- A possible organization is stored in the fig below...
- 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.
- The cells in each column are connected to sense/write circuit by two bit lines.
- The sense /write circuits are connected to the data input/output lines of the chip.
- During read operation these circuits sense or read the information stored in cells selected by a word line and transmit the information to the output lines.
- During write operation the sense/write circuit receives the input information and store in the cell of selected word.

### **Types of Memory:**

There are two types of memory chips

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

#### **a) ROM Chips:**

ROM chips are designed for applications in which data is read. These chips are programmed with data by an external programming unit before they are added to the computer system. Once it is done the data does not change. A ROM chip always retains its data, even when

Power to chip is turned off so ROM is called **nonvolatile** because of its property. There are several types of ROM chips which are differentiated by how often they are programmed.

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

Once chip is designed the data will not change. Figure below shows the possible configuration of the ROM cell.

- Logic 0 is stored in the cell if the transistor is connected to ground at point P, otherwise 1 stored.
- A sense circuit at the end of the bit line generates a high voltage indicating a 1. Data are written into the ROM when it is manufactured.

### **PROM**

- Some ROM designs allow the data to be loaded by the user, thus providing programmable ROM (PROM).
- Programmability is achieved by inserting a fuse at point P in the above fig. Before it is programmed, the memory contains all 0's.
- The user inserts 1's at the required locations by burning out the fuse at these locations using high current pulse.
- The fuses in PROM cannot be restored once they are blown, PROM's can only be programmed once.

### **2) EPROM**

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

- Programming in 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 being placed under UV light, which causes the capacitor to leak their charge.

### 3) 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.

### 4) 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.

## 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 digital circuit in which it is used stores data at various locations in the RAM are retrieves data from these locations.
- 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)

### 1. 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.
- DRAM memory cell is shown in the figure below.

## 2. 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.

➤ Various factors such as cost, speed, power consumption and size of the chip determine how a RAM is chosen for a given application

➤ Static RAMs:

- Chosen when speed is the primary concern.
- Circuit implementing the basic cell is highly complex, so cost and size are affected.
- Used mostly in cache memories.

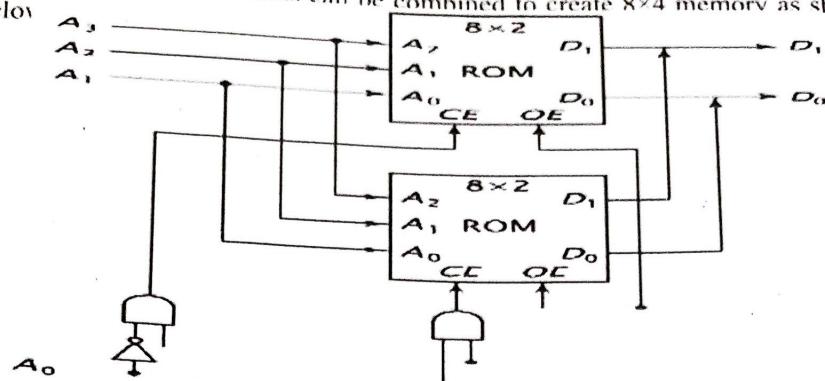
➤ Dynamic RAMs:

- Predominantly used for implementing computer main memories.
- High densities available in these chips.
- Economically viable for implementing large memories

## Memory subsystem configuration:

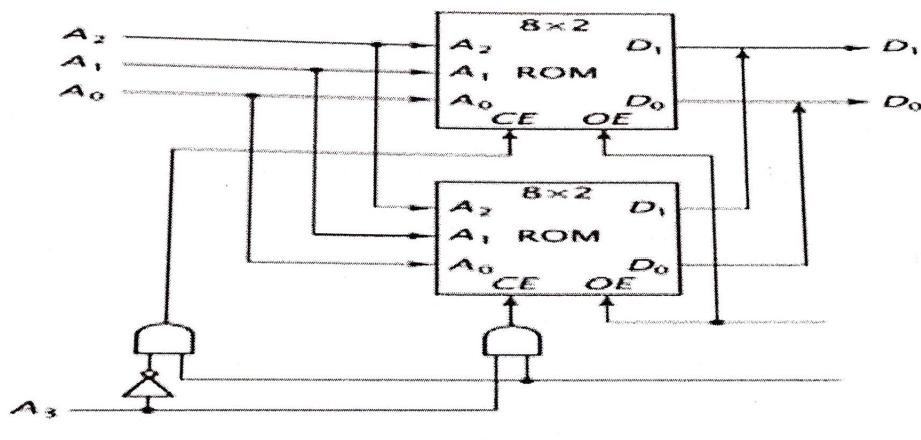
- Two or more chips are combined to create memory with more bits per location. This is done by connecting the corresponding address and control signals of chips and connecting their data pins to different bits of the data bus.
- The following figure followed the higher order interleaving where the higher order bit A3 is 0 and lower chip has A3 1. This difference is used to select one of the two chips.
- In the lower order interleaving we consider lower bit for selecting the two chips. So the upper chip is enabled when A0=0 or by address 0×××(or 0,2,4,6,8,10,12) and lower chip is enabled when A0=1 which is true for the address 1,3,5,7,9,11,13 and 15. Figure for lower order interleaving is shown in the figure below.

- For example two  $8 \times 2$  chips can be combined to create  $8 \times 4$  memory as shown the figure below



(b)  
An  $8 \times 4$  memory subsystem constructed from two  $8 \times 2$  ROM chips

- Both receive the same address inputs from the bus as well as the same chips enable and output enable signals. The data pins of the first two connected to bits 3 and 2 of the data bus and other chip are connected to bits 1 and 0.
- When the CPU reads the data it places the address on the address bus. Both the chips read the address and perform their internal decoding.
- If CE (Chip Enable) and OE (Output Enable) signals are activated. The chips place the data on the data bus because OE and CE are same for both chips. The same  $8 \times 2$  configured as a  $16 \times 2$  shown in the figure below.



(a)  
A  $16 \times 4$  memory subsystem constructed from two  $8 \times 2$  ROM chips with higher order interleaving

- The upper chip is configured as memory locations 0-7(0000 to 0111). The lower chip is configured as 8 to 15(1000 to 1111).
- The configurations can be done by using two methods **higher order interleaving** and **lower order interleaving**.

### Multi byte organization:

- Many data formats use more than one 8-bit byte to represent a value whether it is an integer, floating point, or character string.
- Most CPU assign addresses to 8-bit memory locations so these values must be stored in more than one location. It is necessary for every CPU to define the order it expects for the data in these locations.
- 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.

### **1.4 I/O SUBSYSTEM ORGANIZATION AND INTERFACING**

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

Below figure shows the basic connection between the CPU, memory to the I/O device. The I/O device is connected to the computer system address, data and control buses. Each I/O device includes I/O circuitry that interacts with the buses.

#### **INPUT DEVICE:**

- The generic interface circuitry for an input device such as keyboard and also enable logic for tri state buffer is shown in the figure below.

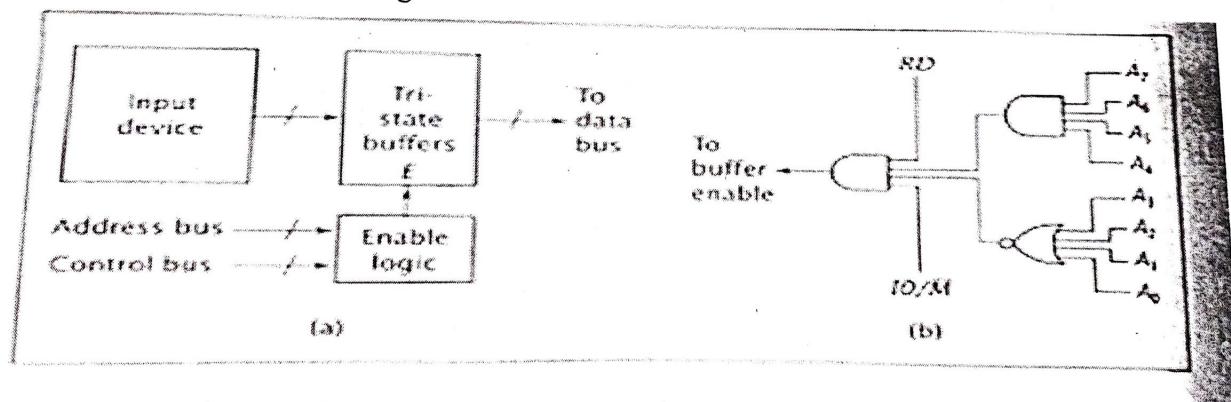
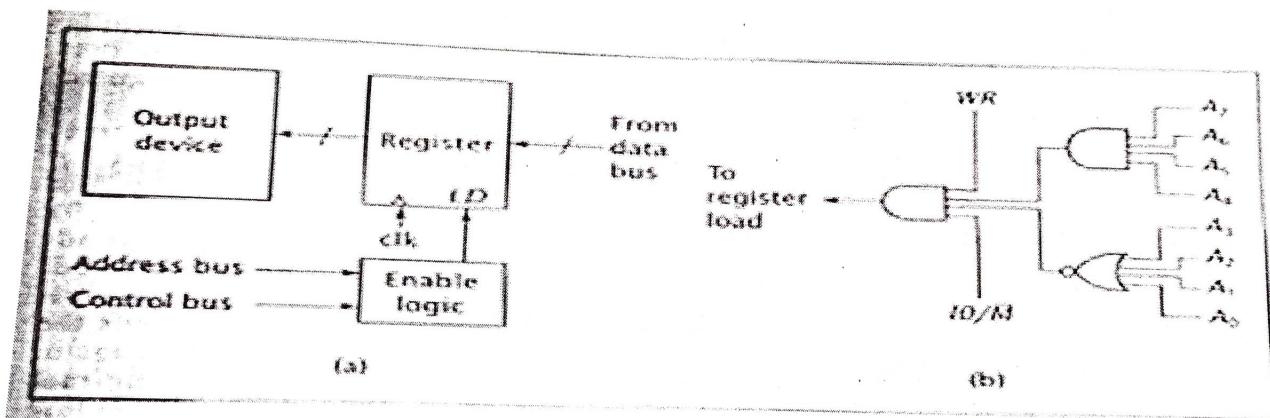


Fig 1.5: (a) with its interface and (b) the enable logic for the tri-state buffers

- 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 IO/.

### 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.
- The design of the interface circuitry for an output device, such as a computer monitor, is somewhat different than that 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 than 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.
  - When the load logic receives the correct address and control signals, it asserts data bus. The output device can read the data from the register at its leisure while the CPU performs the other tasks.



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 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.

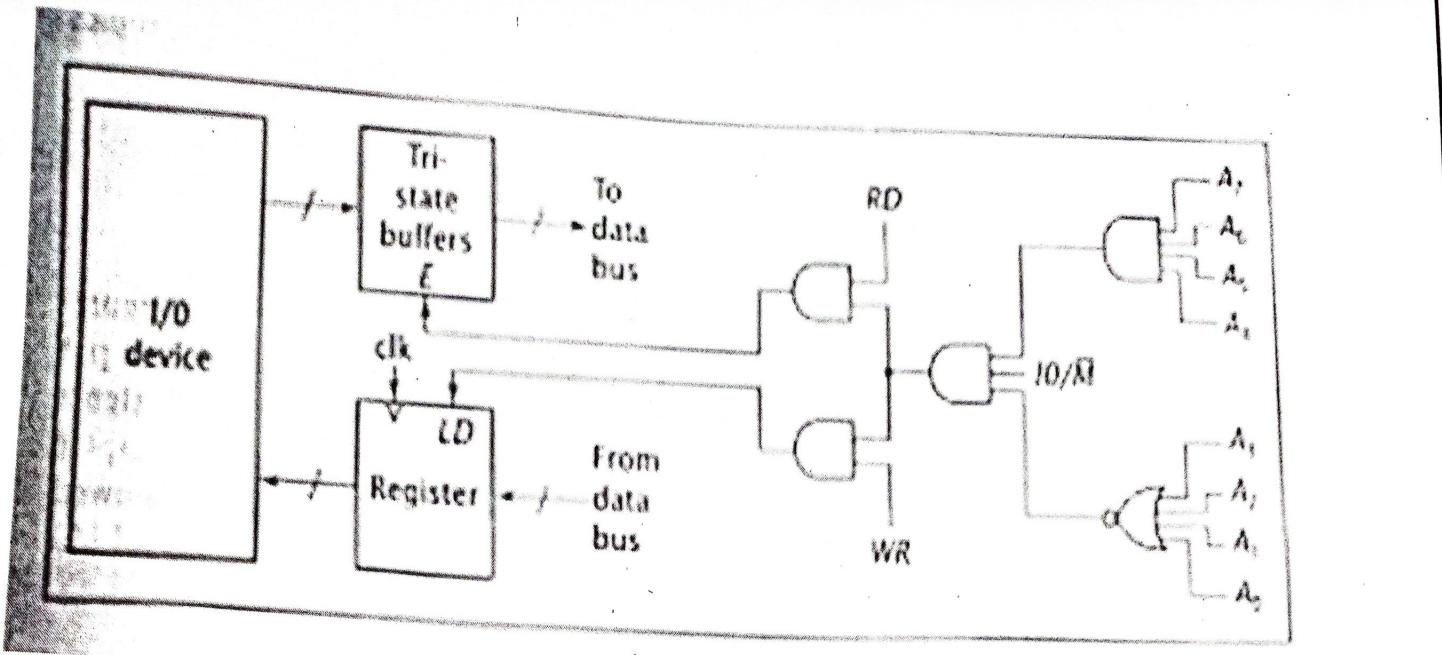
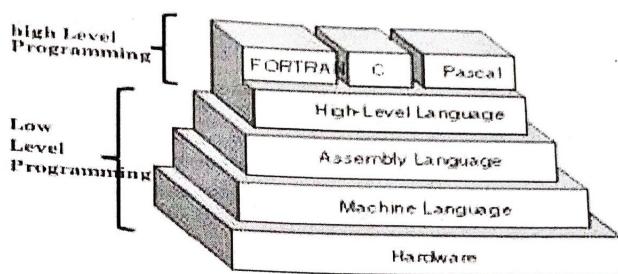


Fig: A bidirectional I/O device with its interface and enable/load logic

### 1.5 LEVELS OF PROGRAMMING LANGUAGES

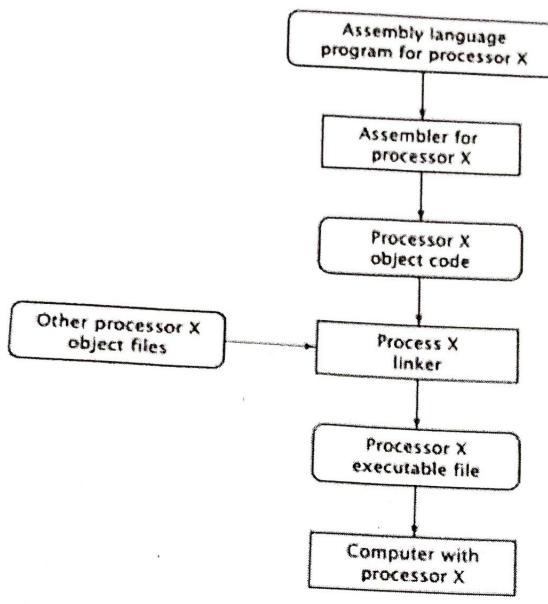
- Computer programming languages are divided into 3 categories.
  - High level language
  - Assembly level language
  - Machine level language
- **High level languages** are platform independent that is these programs can run on computers with different microprocessor and operating systems without modifications. 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.
- A program written in the assembly language of one processor cannot be run on a computer that has different processor so assembly languages are platform dependent.
- Unlike high level languages, instructions in assembly languages can directly manipulate the data stored in microprocessor internal components. Assembly language instructions can load the data from memory into microprocessor registers, add values, and perform many other operations.
- 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.

- Levels of programming languages is shown in the figure below



- Programmers don't written the programs in machine language ,rather programs written in assembly or high level are converted into machine level and then executed by the 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**.
- The process of converting the assembly language program to an executable form is shown the figure below.

# 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

**Fig: Assembly process for assembly language programs**

## 1.6 ASSEMBLY LANGUAGE INSTRUCTIONS:

- The simplest way to organize a computer is to have one processor register and instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.
- Below Figure depicts this type of organization. Instructions are stored in one section of memory and data in another.
- For a memory unit with 4096 words we need 12 bits to specify an address since  $2^{12} = 4096$ . If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated op code) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit

address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

**a) Instruction formats:**

- The basic computer has three instruction code formats, as shown in Fig below. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A memory reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address.
- The register-reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an

Operation on or a test of the AC register. An operand from memory is not needed therefore the other 12 bits are used to specify the operation or test to be executed.

- Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed. The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three op code bits in positions 12 through 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I. If the 3-bit op code is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an I/O type.
- The instruction for the computer is shown the table below. The symbol designation is a three letter word and represents an abbreviation intended for programmers and users. The hexa decimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits.
- A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When I = 1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is I.
- Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits.
- The input-output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Sym	Hexadecimal code	Description
AND	0xx	AND memory word to AC
ADD	1xx	ADD memory word to AC
LDA	2xx	Load memory word to AC
STA	3xx	Store content of AC in
BUN	4xx	Branch Unconditionally
BSA	5xx	Branch and save return
ISZ	6xx	Increment & skip if skip
CLA		
CLE	7800	Clear AC
CMA	7400	Clear E
CME	7200	Complement AC
CIR	7100	Complement E
CIL	7080	Circulate right AC & E
INC	7040	Circulate left AC & E
SPA	7020	Increment AC
SNA	7010	Skip next address if AC is
SZA	7008	Skip next address if AC is -
SZE	7004	Skip next address if AC is
HLT	7002	Skip next address if E is zero
INP	7001	Halt computer
OUT	F800	Input character to AC
SKI	F400	output character from AC
SKO	F200	Skip on input flag
ION	F100	Skip on output flag
IOF	F080	Interrupt on
	F040	Interrupt off

Fig: Basic computer instructions

### Instruction types:

- Assembly languages instructions are grouped together based on the operation they performed.
  - Data transfer instructions
  - Data operational instructions
  - Program control instructions

#### 1) 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.

- **Output the data from the microprocessor:** The microprocessor copies the data from one of the registers to an input device such as digital display of a microwave oven.

2) 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.

3) 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.
- A microprocessor can be designed to accept interrupts. An interrupt causes the processor to stop what is doing and start other instructions. Interrupts may be software or hardware.
- One final type of control instructions is halt instruction. This instruction causes a processor to stop executing instructions such as end of a program.

### 1.7 Instruction set architecture(ISA) design:

- Designing of the instruction set is 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.
- If the processor is to be used for general purpose computing such as personal computer it will probably require a rich set of ISA. It will need a relatively large instruction set to

- perform the wide variety of tasks it must accomplish. It may also need many registers for some of these tasks.

- In contrast consider a microwave oven it require only simple ISA those need to control the oven. This 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.
- 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 from the memory.
- Having too few registers causes a program to make more reference to the memory thus reducing performance.
- General purposes CPU have many registers since they have to process different types of programs. Intel processor has 128 general purpose registers for integer data and another 128 for floating point data. For dedicated CPU such as microwave oven having too many registers adds unnecessary hardware to the CPU without providing any benefits.

## 1.7 A RELATIVELY SIMPLE INSTRUCTION SET ARCHITECTURE:

- A relatively simple instruction set architecture describes the ISA of the simple processor or CPU.
- A simple microprocessor can access  $64K (=2^{16})$  bytes of memory with each byte having 8 bits or  $64K \times 8$  of memory. This does not mean that every computer constructed using this relatively simple CPU must have full 64K of memory. A system based on this processor can have less than memory if doesn't need the maximum 64K of memory.
- 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.
  - Isolated I/O
  - Memory mapped I/O
- 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 ( $Z$ )

- The first accumulator is an 8bit register. Accumulator of this CPU receives the result of instructions that use two operands. It also provides one of the operands for ALU accumulator and also data stored from memory it is loaded into the
- Register R is an 8-bit general purpose register. It supplies the second operand of all two operand arithmetic and logical instructions. It also stores the final data.
- Finally, there is a 1-bit zero flag  $Z$ . Whenever an arithmetic and logical instruction is executed. If the result of the instruction is 0 then  $Z$  is set to 1 that a zero result was generated. Otherwise it is set to 0.
- The final component is the instruction set architecture for this relatively simple CPU is shown in the table below.

Instruction	Code	Instruction	Operation
NOP	0000 0000		No operation
LDAC	0000 0001 $\Gamma$	$AC = M[\Gamma]$	
STAC	0000 0010 $\Gamma$	$M[\Gamma] = AC$	
MVAC	0000 0011	$R = AC$	
MOVR	0000 0100	$AC = R$	
JUMP	0000 0101 $\Gamma$	GOTO $\Gamma$	
JMPZ	0000 0110 $\Gamma$	IF ( $Z=1$ ) THEN GOTO $\Gamma$	
JPNZ	0000 0111 $\Gamma$	IF ( $Z=0$ ) THEN GOTO $\Gamma$	
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 $Z = 0$	

- The LDAC, STAC, JUMP, JMPZ AND JPNZ instructions all require a 16-bit memory address represented by the symbol  $\Gamma$ .
- 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.

- The instructions of this instruction set architecture can be grouped into 3 categories
  - Data transfer instructions
  - Program control instructions
  - Data operational instructions
- The NOP, LDAC, STAC, MVAC or MOVR instructions are **data transfer instructions**.  
The NOP operation performs no operation. The LDAC operation loads the data from the memory it reads the data from the memory location  $M[\Gamma]$ . The STAC performs opposite, copying data from AC to the memory location  $\Gamma$ .
- The MOVAC instruction copies data from R to AC and MOVR instruction copies the data from R to AC.
- There are three **program control instructions** in the instruction set: JUMP, JUPZ and JPNZ. The JUMP is the unconditional it always jumps to the memory location  $\Gamma$ . For example JUMP 1234H instruction always jump to the memory location 1234H. The JUMP instruction uses the immediate addressing mode since the jump address is specified in the instruction. The other two program control instructions JUPZ and JPNZ are conditional. If their conditions are met  $Z=0$  for JUPZ and  $Z=1$  for JPNZ these instructions jump to the memory location  $\Gamma$ .
- Finally data operations instructions are ADD, SUB, INAC and CLAC instructions are arithmetic instructions. The ADD instructions add the content of AC and R and store the result again in AC. The instruction also sets the zero flag  $Z=1$  if sum is zero or  $Z=0$  if the sum is non zero value. The SUB operation performs the same but it subtracts the AC and R.
- INAC instruction adds 1 to the AC and sets Z to its proper value. The CLAC instruction always makes  $Z=1$  because it clears the AC value that is AC to 0.
- The last four data operation instructions are logical instructions as the name implies the AND, OR, and XOR instructions logically AND, OR and XOR the values AC and R and store the result in AC. The NOT instruction sets AC to its bitwise complement.

## 1.8 RELATIVELY SIMPLE COMPUTER:

- In this relatively simple computer we put all the hardware components of the computer together in one system. This computer will have 8K ROM starting at address 0 followed 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 M. The relatively simple computer is shown in the figure below. It only contains the CPU details. Each part will be developed in the design.

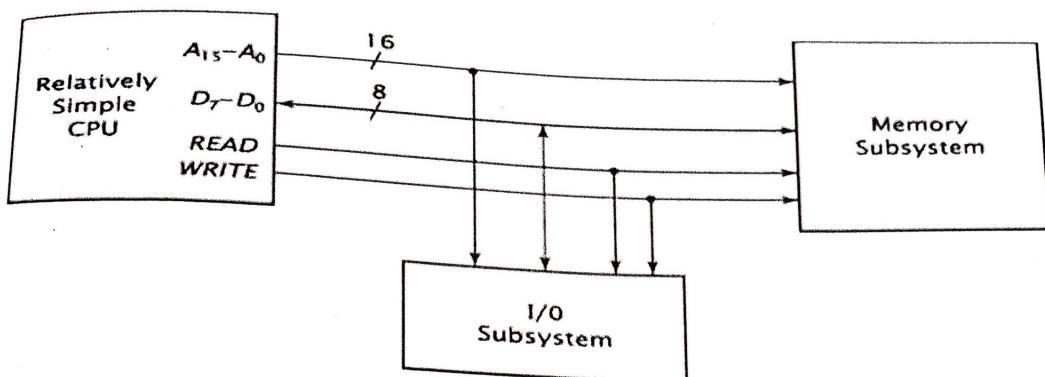


Fig: A relatively simple computer: CPU details only

- To access a memory chip the processor must supply an address used by the chip. An 8K memory chip has  $2^{13}$  internal memory locations it has 13bit address input to select one of these locations. The address input of each chip receives CPU address bits A<sub>12</sub> to A<sub>0</sub> from system address bus. The remaining three bits A<sub>15</sub>, A<sub>14</sub>, and A<sub>13</sub> will be used to select one of the memory chips.
- The address range of ROM and RAM is

## Unit-2

### ORGANIZATION OF A COMPUTER

Register transfer: Register transfer language, register transfer, bus and memory transfers, arithmetic micro operations, logic micro operations, And shift micro operations; Control unit: Control memory, address sequencing, micro program example, and design of control unit.

#### Register Transfer and Microoperations

##### 2.1 Register Transfer Language

- A digital system is an interconnection of digital hardware module. Digital systems varies in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers.
- Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic.
- The various modules are interconnected with common data and control paths to form a digital computer system.
- Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations.
- A microoperation is an elementary operation performed on the information stored in one or more registers.
- The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load.

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

1. The set of registers it contains and their function.
  2. The sequence of microoperations performed on the binary information stored in the registers.
  3. The control that initiates the sequence of microoperations.
- The symbolic notation used to describe the microoperation transfers among registers is called a **register transfer language**.
  - The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register.
  - The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process.
  - A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module.
  - It is a convenient tool for describing the internal organization of digital computers in

concise and precise manner.

It can also be used to facilitate the design process of digital systems.

- The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize.
- To define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations.
- Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

## 2.2 Register Transfer

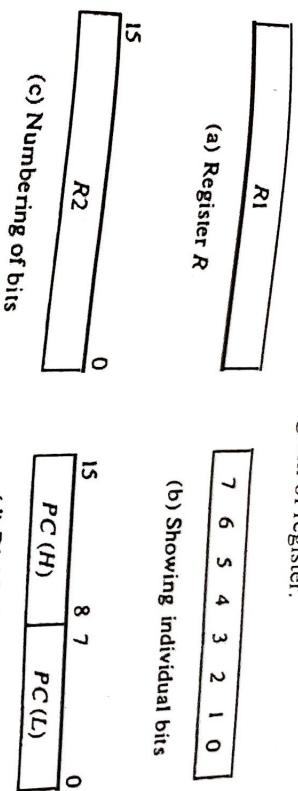
- Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, MAR,PC,IR. The individual flip-flops
- Figure 1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 1(a) the individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC(L) refers to the low-order byte and PC(S-15) or PC(H) to the high-order byte.
- Information transfer from one register to another is designated in symbolic form by means of a replacement operator.
- The statement  $R2 \leftarrow R1$  denotes a transfer of the content of register R1 into register R2.
- It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.
- A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability.
- The transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

If ( $P = 1$ ) then ( $R2 \leftarrow R1$ )

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

P:  $R2 \leftarrow R1$

Figure 1 Block diagram of register.



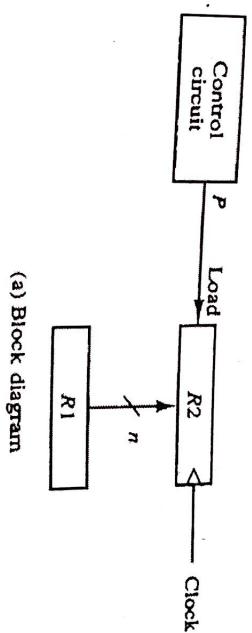
The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if  $P = 1$ .

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.

- Above figure shows the block diagram that depicts the transfer from R1 to R2. The n outputs of register R1 are connected to the n inputs of register R2.
- The basic symbols of the register transfer notation are listed in Table 4-1. Registers are denoted by capital letters, and numerals may follow the letters.

$$T: R2 \leftarrow R1, R1 \leftarrow R2$$

The statement denotes an operation that exchanges the contents of two registers during one common clock pulse provided that  $T = 1$ . This simultaneous operation is possible with registers that have edge-triggered flip-flops.



(a) Block diagram

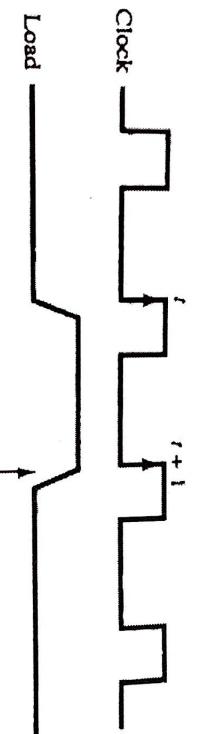


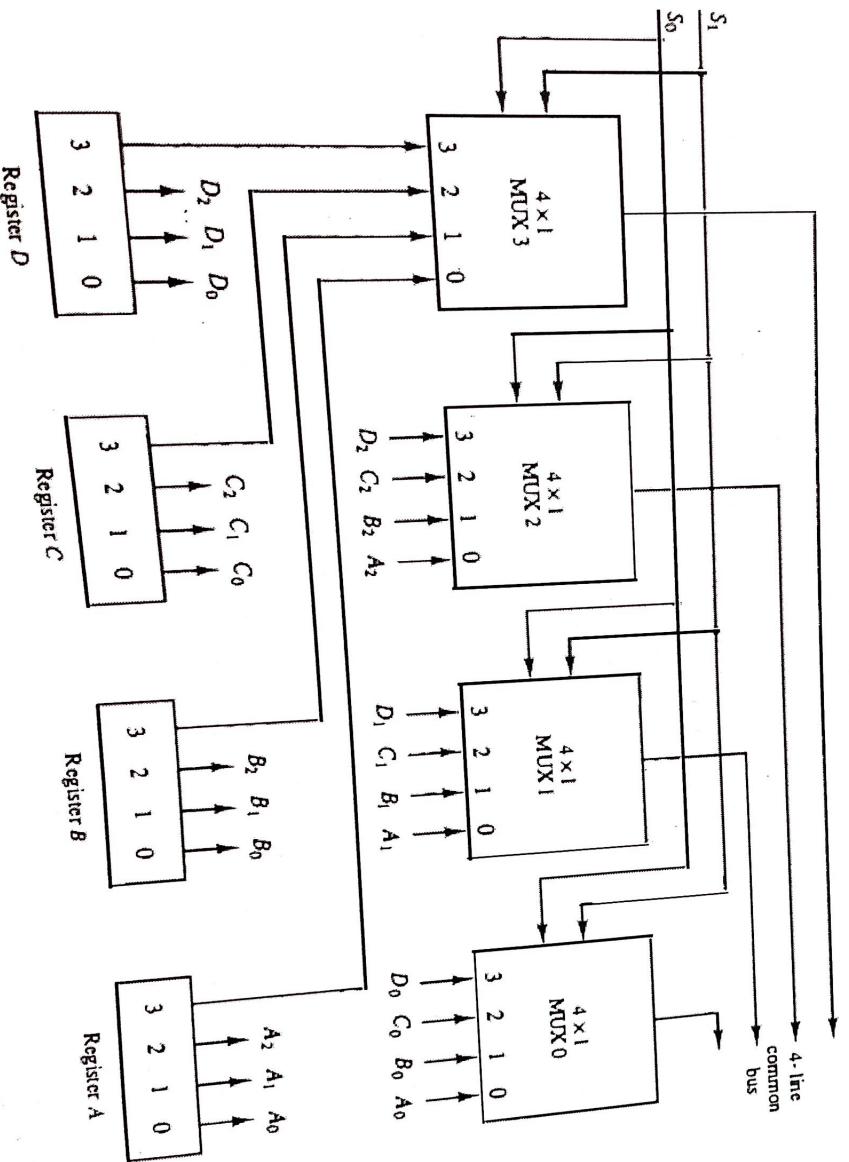
TABLE 1 Basic symbols for register Transfers

Symbol	Description	Examples
Letters and numerals	Denotes a register	MAR,R2
Parenthesis()	Denotes a part of registers	R290-70,R2(L)
Arrow	Denotes transfer of information	R2 $\leftarrow$ R1
Comma ,	Separates two microoperations	R2 $\leftarrow$ R1, R $\leftarrow$ R2

### 2.3 Bus and Memory Transfers

- A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system.
- A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system.
- A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.
- The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus.
- MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

Figure 3: Bus systems for four registers



- The two selection lines  $S_1$  and  $S_0$  are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus.
  - When  $S_1S_0 = 00$ , the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus.
  - This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if  $S_1S_0 = 01$ , and so on.
- Table 2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

Table 2 function table for Bus of fig 3

$S_1$	$S_0$	Registers selected
0	0	A
0	1	B
1	0	C
1	1	D

- In general, a bus system will multiplex k registers of n bits each to produce an n-line common bus.
- The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be  $k \times 1$  since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus.
- Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.
- The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected.
- The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is included in the statement, the register transfer is symbolized as follows:

$\text{BUS} \blacktriangleright C,$

The content of register C is placed on the bus, and the content of the bus is loaded into register R

1 by activating its load control input.

If the bus is known to exist in the system, it may be convenient just to show the direct transfer.  $R1 \leftarrow C$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

### Three state bus buffers

- A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state.
- The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance. Three-state gates may perform any conventional logic, such as AND or NAND.
- The graphic symbol of a three-state buffer gate is shown in Fig. 4. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state.

The construction of a bus system with three-state buffers is demonstrated in Fig. 5. To construct a common bus for four registers of  $n$  bits each using three-state buffers, we need  $n$  circuits with four buffers in each as shown in Fig. 5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of  $n$  lines. Only one decoder is necessary to select between the four registers.

Fig 4: Graphic symbol for 3 state buffers

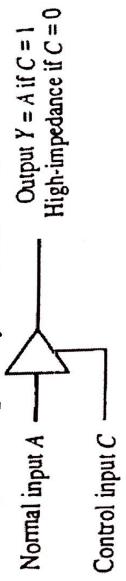
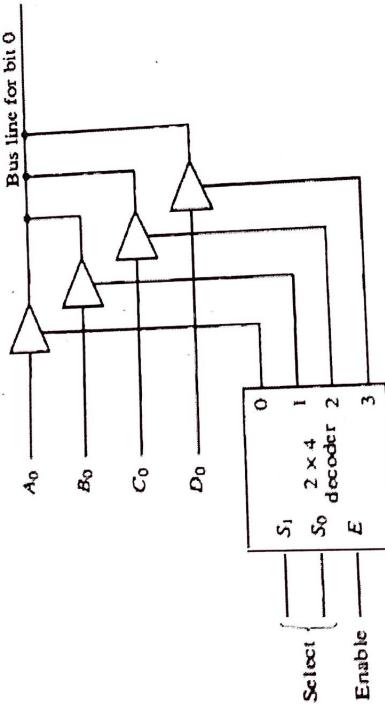


Fig 5: Bus line with 3 state buffer



### Memory Transfer

- The operation of a memory unit was described in Sec. 2-7. The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation.
- A memory word will be symbolized by the letter  $M$ . The particular memory word among the many available is selected by the memory address during the transfer. It is necessary

to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M. Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

Read:  $DR \leftarrow [AR]$

This causes a transfer of information into DR from the memory word M selected by the address in AR. The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R 1

$R_3 \leftarrow R_1 + \overline{R_2} + 1$

$R_2$  is the symbol for the 1's complement of  $R_2$ . Adding 1 to the 1's complement produces the 2's complement of  $R_2$ . Adding 1 to the 2's complement of  $R_2$  is equivalent to  $R_1 - R_2$ .

TABLE 3 Arithmetic Microoperations

Symbolic designation	Description
$R_3 \leftarrow R_1 + R_2$	Contents of $R_1$ plus $R_2$ transferred to $R_3$
$R_3 \leftarrow R_1 - R_2$	Contents of $R_1$ minus $R_2$ transferred to $R_3$
$R_2 \leftarrow \overline{R_2}$	Complement the contents of $R_2$ (1's complement)
$R_2 \leftarrow \overline{R_2} + 1$	2's complement the contents of $R_2$ (negate)
$R_3 \leftarrow R_1 + \overline{R_2} + 1$	$R_1$ plus the 2's complement of $R_2$ (subtraction)
$R_1 \leftarrow R_1 + 1$	Increment the contents of $R_1$ by one
$R_1 \leftarrow R_1 - 1$	Decrement the contents of $R_1$ by one

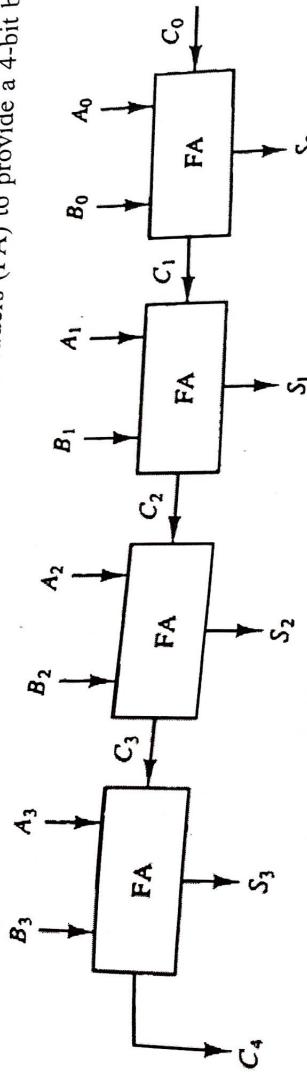
- The increment and decrement microoperations are symbolized by plus one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.
- The arithmetic operations of multiply and divide are not listed in Table 3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations.
- The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit.

## 2.4 Arithmetic micro operations

### Binary Adder

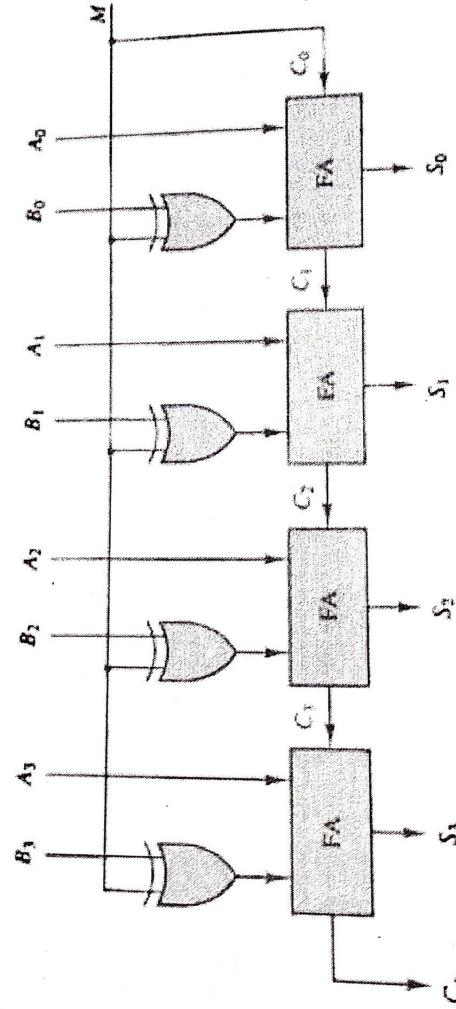
- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder.

- 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 from one full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder.



**Binary Adder-Subtractor**

- The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-2. Remember that the subtraction  $A - B$  can be done by taking the 2's complement of  $B$  and adding it to  $A$ .
- The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the sum carry.
- The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 4-7.



**Fig7: 4 bit adder- subtractor**

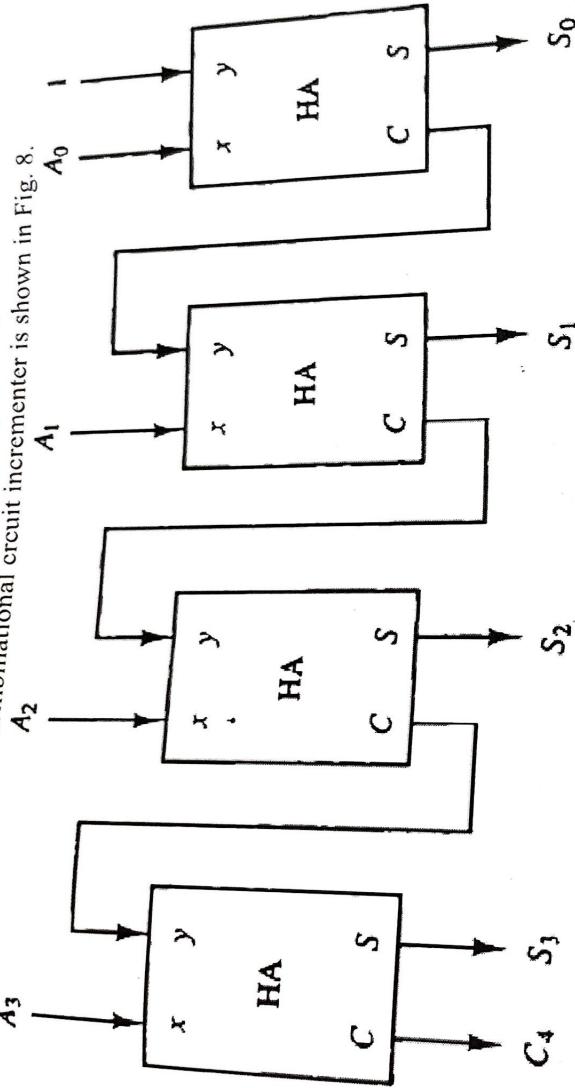
### Binary Incrementer

- The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented.

- This microoperation is easily implemented with a binary counter every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. Binary incrementer.

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. The diagram of a 4-bit combination incrementer.

Combinational circuit incrementer is shown in Fig. 8.



A withmosti 2

Fig 8: 4- bit binary incrementer

- The arithmetic microoperations listed in Table 4-3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder.
  - By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations. The diagram of a 4-bit arithmetic circuit is shown in Fig. 9. The output of the binary adder is calculated from the following arithmetic sum:

$$D = A + Y + C \text{ in}$$

Where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C<sub>in</sub> is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S1 and S0 and making C<sub>in</sub> equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4.

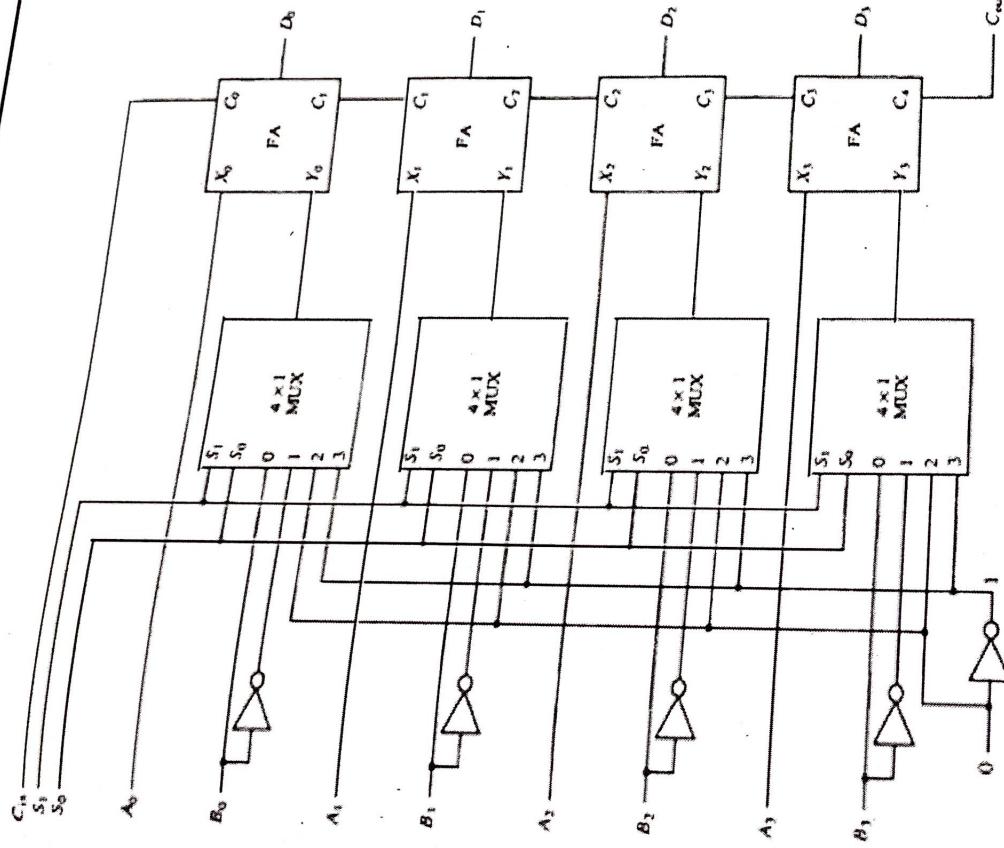


Fig 9: 4-bit arithmetic circuit

TABLE 4 Arithmetic Circuit Function Table

Select	$S_1$	$S_0$	$C_{in}$	Input $Y$	Output $D = A + Y + C_{in}$	Microoperation
0	0	0	0	$B$	$D = A + B$	Add
0	0	0	1	$\bar{B}$	$D = A + B + 1$	Add with carry
0	1	0	0	$\bar{B}$	$D = A + \bar{B}$	Subtract with borrow
0	1	1	0	$\bar{B}$	$D = A + \bar{B} + 1$	Subtract
1	0	0	0	0	$D = A$	Transfer $A$
1	0	1	0	0	$D = A + 1$	Increment $A$
1	1	0	1	1	$D = A - 1$	Decrement $A$
1	1	1	1	1	$D = A$	Transfer $A$

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R<sub>1</sub> and R<sub>2</sub> is symbolized by the statement

$$P \cdot R_1 + R_2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of R<sub>1</sub> be 1010 and the content of R<sub>2</sub> be 1 100. The exclusive-OR microoperation stated above symbolizes the following logic computation.

$$1010 \text{ Content of } R_1$$

$$1100 \text{ Content of } R_2$$

$$0110 \text{ Content of } R_1 \text{ after } P = 1$$

The content of R<sub>1</sub>, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R<sub>2</sub> and previous values of R<sub>1</sub>. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

- Special symbols will be adopted for the logic microoperations OR, AND, and Boolean functions.
- The symbol V will be used to denote an OR microoperation and the symbol 1\ to denote complement, to distinguish them from the corresponding symbols used to express AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name.
  - By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs.

- P + Q; R 1 R2 + R3, R4 R5 V R6 List of

### Logic Microoperations

- There are 16 different logic operations that can be performed with two binary variables.
- They can be determined from all possible truth tables obtained with two binary variables as shown in Table 5.
- In this table, each of the 16 columns F0 through F15 represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.
- The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B

TABLE 5 Truth Tables for 16 Functions of Two Variables

$x$	$y$	$F_0$	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	$F_8$	$F_9$	$F_{10}$	$F_{11}$	$F_{12}$	$F_{13}$	$F_{14}$	$F_{15}$
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	1	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

TABLE 6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \bar{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \bar{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \bar{A} \vee \bar{B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \bar{A} \oplus \bar{B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \bar{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \bar{B}$	
$F_{12} = x'$	$F \leftarrow \bar{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \bar{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \bar{A} \wedge \bar{B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all } 1\text{'s}$	Set to all 1's

### Hardware Implementation

- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function.
- Although there are 16 logic microoperations, most computers use only four-AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.
- Figure 1.0 shows one stage of a circuit that generates the four basic logic microoperations.
- It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer.

### Some Applications

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B). In a typical

application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

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's in B. The following numerical example clarifies this operation:

1010 A before	
1100 B (logic operand)	
1110 A after	

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register.

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

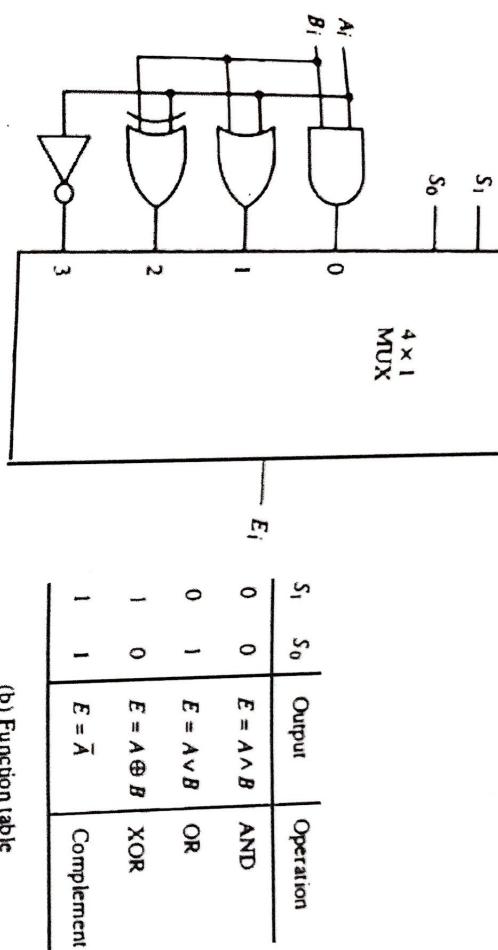
1010 A before	
1100 B (logic operand)	
0110 A after	

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register. The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

1010 A before	
1100 B (logic operand)	
0010 A after	

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is  $AB'$ . The corresponding logic microoperation is  $A \wedge B$ .

Figure 10 One stage of logic circuit.



(a) Logic diagram

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 an AND micro operation as seen from the following numerical example:

1010 A before

1100 B (logic operand)

1000 A after masking

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. The two leftmost bits are left unchanged because the corresponding bits of B are 1's. The mask operation is more convenient to use than the selective clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear. The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010.

To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0110 1010 A Before

0000 1111 B(mask)

and then insert the new value:

0000 1010 A before

1001 0000 B (insert)

1001 1010 A after insertion

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

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 an exclusive-OR microoperation as shown by the following example:

1010 A  
1010 B  
0000 A <-A + B

- When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

## 2.5 Shift Microoperations

- Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations
- During a shift-left operation the serial input transfers a bit into the leftmost position. During a shift-right operation the serial input transfers a bit into the rightmost position.
- 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. shl and shr for logical shift-left and shift-right microoperations. For example:  
 $R1 \leftarrow s\_{hl}\ R1$   
 $R2 \leftarrow s\_{shr}\ R2$

are two microoperations that specify a 1-bit shift to the left of the content of register R 1 and a 1-bit shift to the right of the content of register R2. The symbolic notation for the shift microoperations is shown in table 7.

TABLE 7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow shl\ R$	Shift-left register R
$R \leftarrow shr\ R$	Shift-right register R
$R \leftarrow cl\ R$	Circular shift-left register R
$R \leftarrow cir\ R$	Circular shift-right register R
$R \leftarrow ash\ R$	Arithmetic shift-left R
$R \leftarrow ashr\ R$	Arithmetic shift-right R

- An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2.
- An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number.

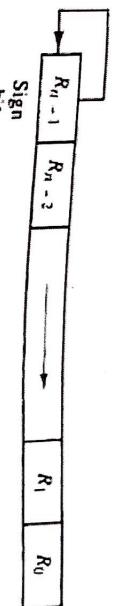


Figure 11 Arithmetic shift right.

If  $V_s = 0$ , there is no overflow, but if  $V_s = 1$ , there is an overflow and a sign reversal after the shift.  $V_s$  must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

$$V_s = K_{n-1} \oplus K_{n-2}$$

#### Hardware Implementation

A possible choice for a shift unit would be a bidirectional shift register with parallel load. Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register. A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 12. The 4-bit shifter has four data inputs,  $A_0$  through  $A_3$ , and four data outputs,  $H_0$  through  $H_3$ . There are two serial inputs, one for shift left (IL) and the other for shift right (IR). When the selection input  $S = 0$ , the input data are shifted right (down in the diagram). When  $S = 1$ , the input data are shifted left (up in the diagram). The function table in Fig. 12 shows which input goes to each output after the shift. A shifter with  $n$  data inputs and outputs requires  $n$  multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

## 2.6 Arithmetic Logic Shift Unit

- The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 13.
- Fig. 13 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables  $P_3, P_2, P_1, P_0$ , and  $C$ . The input carry  $C$ , is used for selecting an arithmetic operation only.

Table 8 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with  $5352 = 00$ . The next four are logic operations and are selected with  $5352 = 01$ .

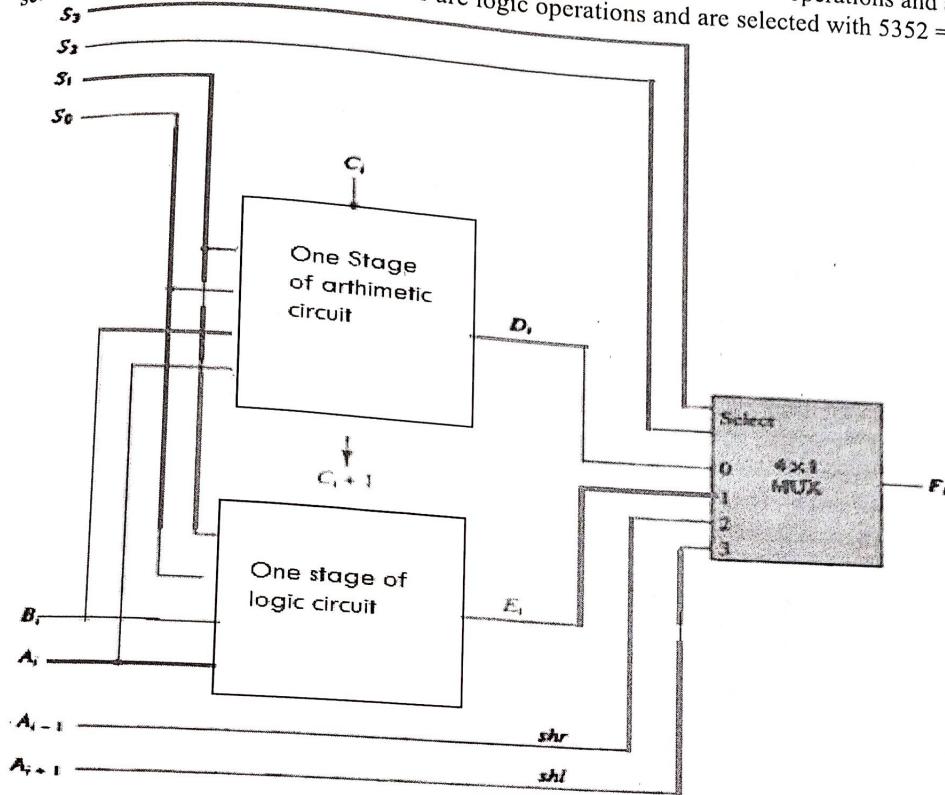


Fig 13: One stage of arithmetic and logic unit

TABLE 8 Function Table for Arithmetic Logic Shift Unit

Operation select						
$S_3$	$S_2$	$S_1$	$S_0$	$C_{in}$	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	$F = A + \bar{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \bar{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement $A$
0	0	1	1	1	$F = A$	Transfer $A$
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	$F = \bar{A}$	Complement $A$
1	0	x	x	x	$F = \text{shr } A$	Shift right $A$ into $F$
1	1	x	x	x	$F = \text{shl } A$	Shift left $A$ into $F$

## 2.7 Control memory

- the control memory can be a read-only memory (ROM). The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM.
- ROM words are made permanent during the hardware production of the unit. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations.
- The content of the word in ROM at a given address specifies a microinstruction. A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory.
- This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a control memory.
- The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Fig. 14.
- The control memory is assumed to be a ROM, within which all control information is permanently stored.
- The control memory address register holds the address of the microinstruction, and the control data register holds the microinstruction read from memory.
- The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address.

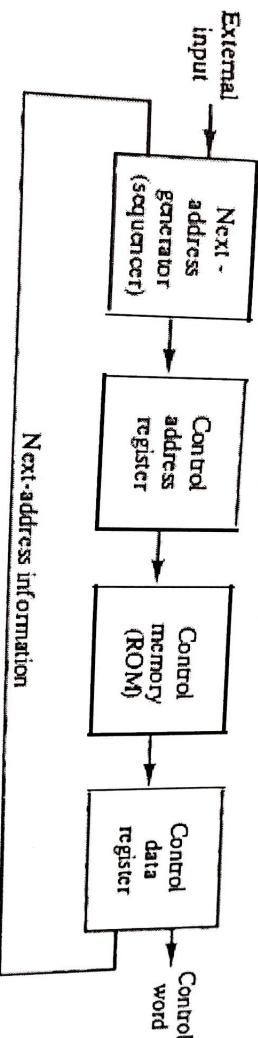


Fig 14: Microprogrammed control organization

- The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

## 2.8 Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a routine. Each computer instruction has its own microprogram routine in control memory to generate the microoperations 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.

- To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction. An initial address is loaded into the control address register when power is turned on in the computer.
- 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 register of the computer.

The next step is to generate the microoperations that execute the instruction fetched from memory.

- The microoperation steps to be generated in processor registers depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process. A mapping procedure is a rule that transforms the instruction code into a control memory address.
- Once the required routine is reached, the microinstructions that execute the instruction sequence of microoperations will depend on values of certain status bits in processor registers.
- Microprograms that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

In summary, 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.

## 2. A facility for subroutine call and return.

Figure 7-2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.

### **Conditional Branching**

The branch logic of Fig. 15 provides decision-making capabilities in the control unit.

The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions..

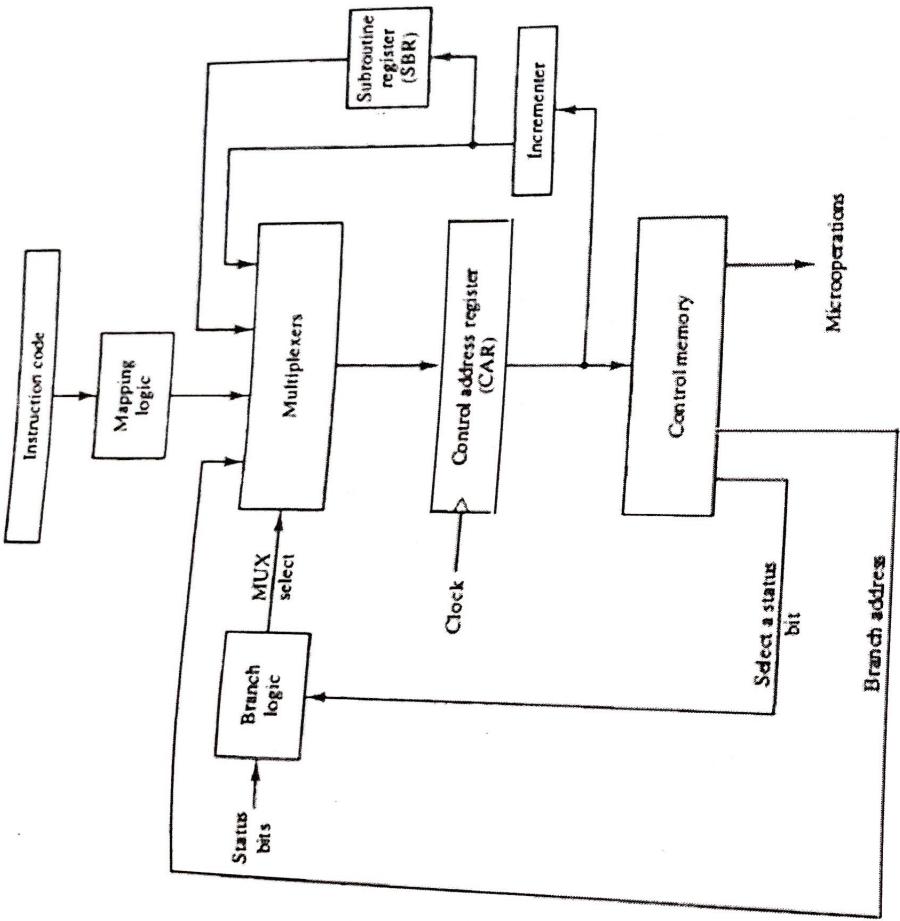
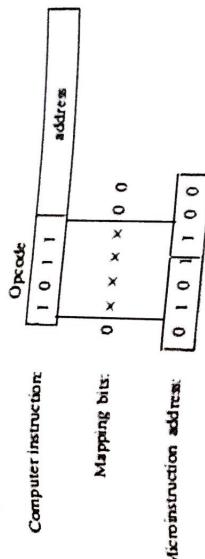


Fig 15: Selection for address for control memory

### Mapping of Instruction

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Fig. 16 has an operation code of four bits which can specify up to 16 distinct instructions.
- Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Fig. 16.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111.



**Fig 16:** mapping of instructions

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

#### **Microprogram Example**

Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming.

#### **Computer Configuration**

The block diagram of the computer is shown in Fig. 17. It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are

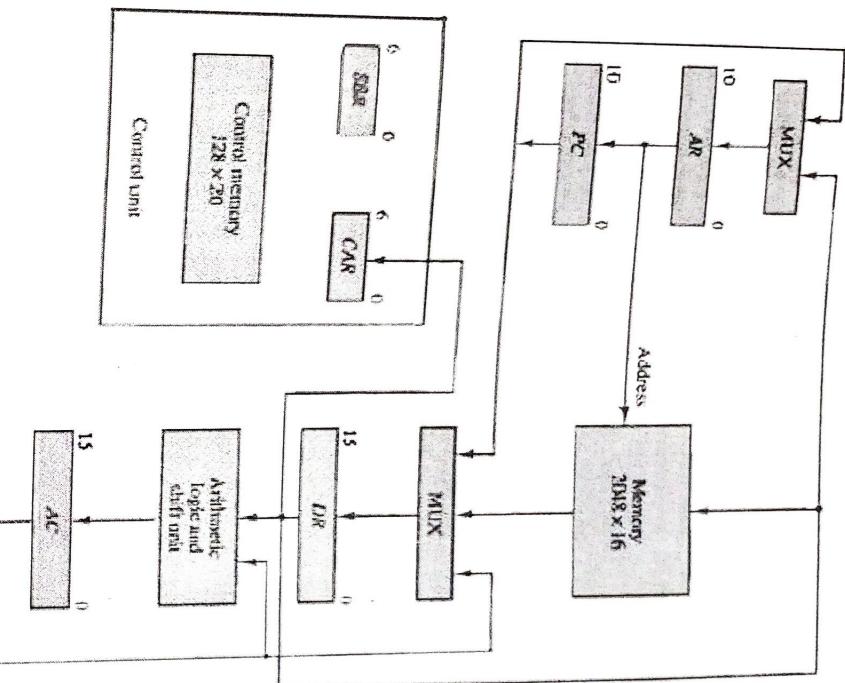


Fig 17 computer hard ware configuration

program counter PC, address register AR, data register DR, and accumulator register AC. The function of these registers is similar to the basic computer.

## 2.8 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  $2^k$  microoperations. Each field requires a decoder to produce the corresponding control signals.
- This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

Figure 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  $3 \times 8$  decoder to provide eight outputs. As shown in Fig. 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. The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.

TABLE 7-4 Input Logic Truth Table for Microprogram Sequence

BR	Input	MUX 1	Load SBR
F <sub>61</sub>	I <sub>1</sub> I <sub>2</sub> T	S <sub>1</sub> S <sub>0</sub>	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 x	1 0	0
1 1	1 1 x	1 1	0

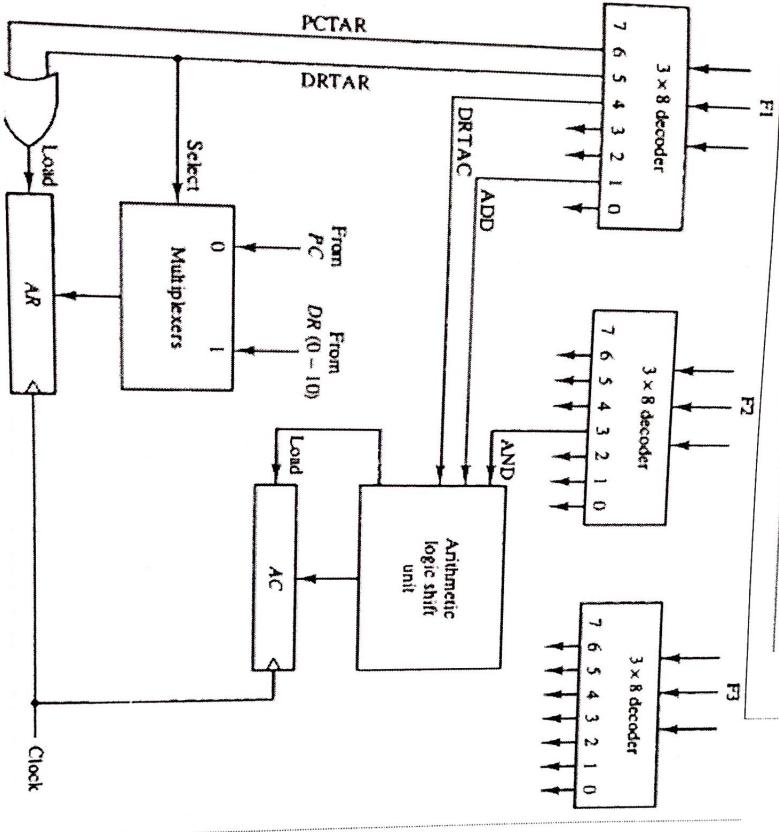


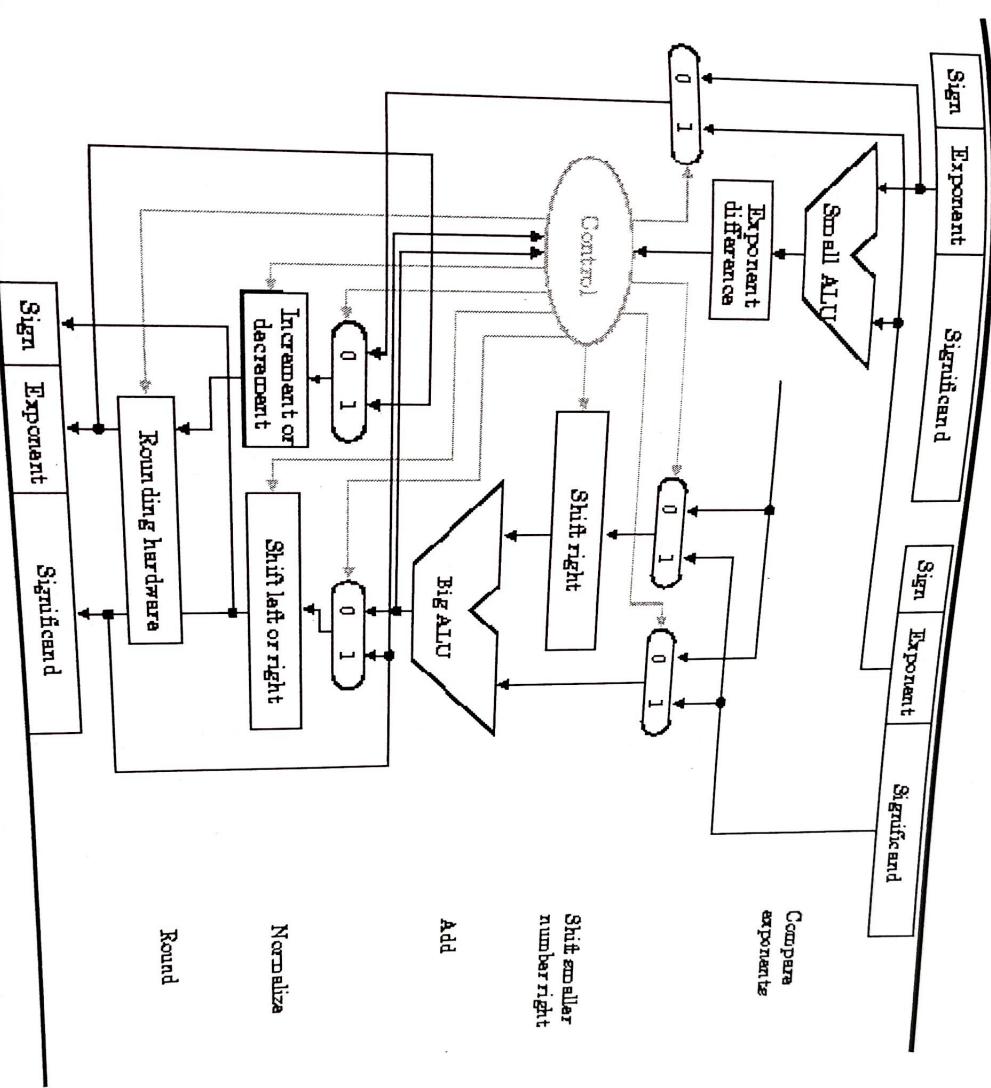
Fig 18: Decoding of micro coding operations

tively, as shown in Fig. 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.

Registers are specified in the .s and .d instructions. Double precision operands are stored in *register pairs* (e.g., \$f0,\$f1 up to \$f30,\$f31). m1 thus handles all the regu-

point operations.  $mfcn$ ,  $mtcn$  instructions are required, while the coprocessor handles the floating point operations.  $mfc0$ ,  $mtc0$ ,  $mfc0$ ,  $mtc0$ , etc.), where  $cn$  refers to the coprocessor(s) and CPU (e.g.,  $mfc0$ ,  $mtc0$ ,  $mfc0$ ,  $mtc0$ , etc.), where  $cn$  refers to the coprocessor(s) and CPU (e.g.,  $mfc0$ ,  $mtc0$ ,  $mfc0$ ,  $mtc0$ , etc.). special I/O operations are required to load and store data between the coprocessor # $n$ . Similarly, (e.g.,  $lwc0$ ,  $swc0$ ,  $lwcl$ ,  $swcl$ , etc.)

FP coprocessors require very complex hardware, as shown in Figure 3.23, which portrays only the hardware required for addition.



**Figure 3.23.** MIPS ALU supporting floating point addition, adapted from [Maf01].

**Figure 3.23.** MIPS ALU supporting –

The use of floating point operations in MIPS assembly code is described in the following simple example, which implements a C program designed to convert Fahrenheit temperatures to Celsius.

```
Float f2c (float fahr) {  
    return ((5.0 / 9.0) * (fahr - 32.0));
```



F2c:

```
lwc1 $f16, const5($gp)      # $f16 = 5.0  
lwc1 $f18, const9($gp)      # $f18 = 9.0  
div.s $f16, $f16, $f18       # $f16 = 5.0/9.0  
lwc1 $f20, const32($gp)     # $f20 = 32.0  
sub.s $f20, $f12, $f20       # $f20 = fahr - 32.0  
mul.s $f0, $f16, $f20       # $f0 = (5/9)*(fahr-32)  
jr   $ra                     # return
```

Here, we assume that there is a coprocessor c1 connected to the CPU. The values 5.0 and 9.0 are respectively loaded into registers \$f16 and \$f18 using the lwc1 instruction with the global pointer as base address and the variables const5 and const9 as offsets. The single precision division operation puts the quotient of 5.0/9.0 into \$f16, and the remainder of the computation is straightforward. As in all MIPS procedure calls, the jr instruction returns control to the address stored in the \$ra register.

## **Memory organization**

### **UNIT-IV**

**memory, cache memory, virtual memory, auxiliary memory, associative interface, asynchronous data transfer; Input or output organization:** Input or output access.

## **MEMORY ORGANIZATION**

- RAM composed of a large number of  $(2M)$  of addressable locations, each of which stores a w-bit word.
- RAM operates as follows: first the address of the target location to be accessed is transferred via the address bus to the RAM's address buffer.
- The address is then processed by the address decoder, which selects the required location in the storage cell unit.
- If a read operation is requested, the contents of the addressed location are transferred from the storage cell unit to the data buffer and from there to the data bus.
- If a write operation is requested, the word to be stored is transferred from the data bus to the selected location in the stored unit. The storage unit is made up of many identical 1-bit memory cells and their interconnections. In each line connected to the storage cell unit, we can expect to
  - find a driver that acts as either an amplifier or a transducer of physical signals.

## **Organization**

- assume that each word is stored in a single track and that each access results in the transfer of a block of words.
- The address of the data to be accessed is applied to the address decoder, whose output determines the track to be used and the location of the desired block of information within the track.
- the track address determines the particular read-write head to be selected. The selected head is moved into position to transfer data to or from the target track. A track position indicator generates the address of the block that is currently passing the read-write head.
- The generated address is compared with the block address produced by the address decoder. The selected head is enabled and the data transfer between the storage track and the memory data buffer register begins.
- The read-write head is disabled when a complete block information has been transferred.

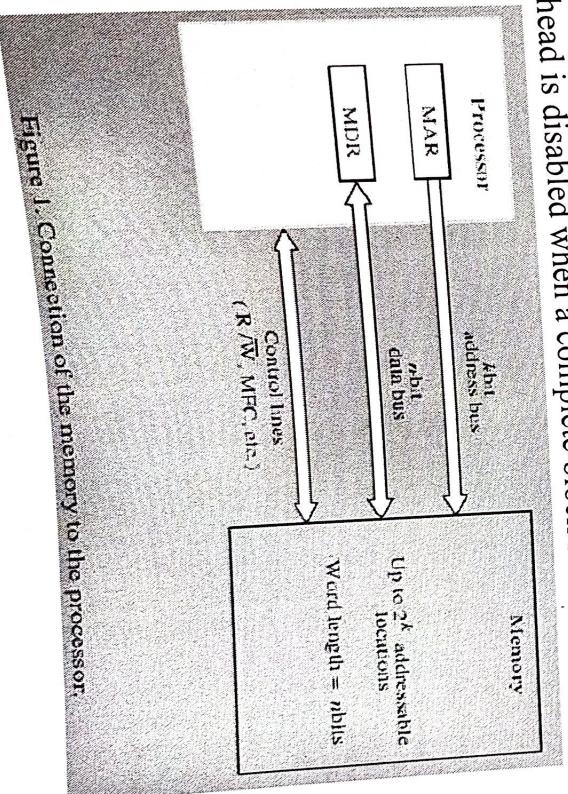


Figure 1. Connection of the memory to the processor.

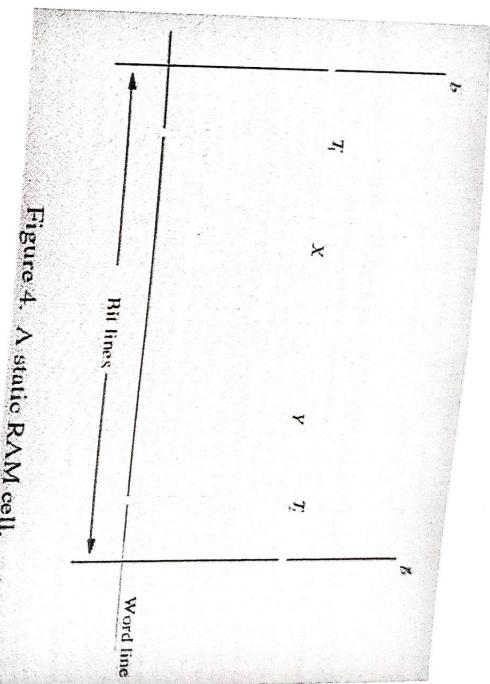


Figure 4. A static RAM cell.

### Static memories (RAM)

- Circuits capable of retaining their state as long as power is applied
- *Static RAM(SRAM)*
- *volatile*

### DRAMs:

<http://csetube.com/>

- Charge on a capacitor
- Needs —Refreshing

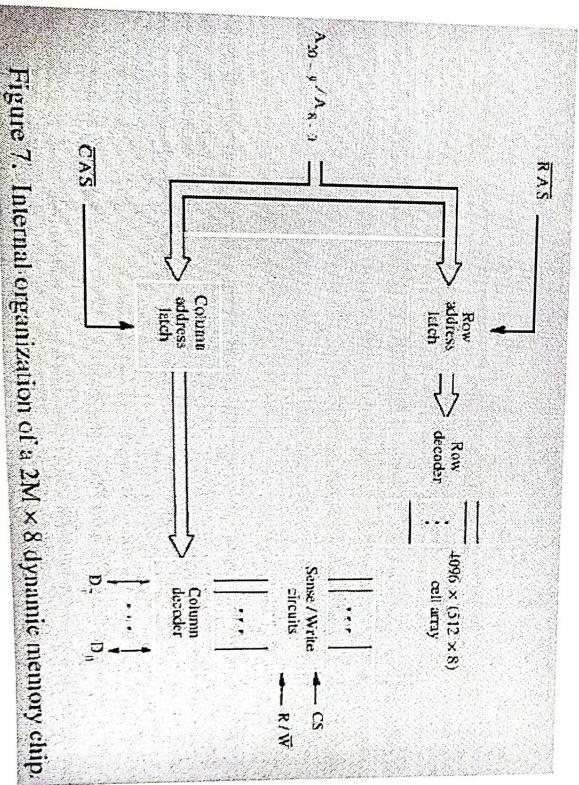


Figure 7. Internal organization of a  $2M \times 8$  dynamic memory chip.

## Synchronous DRAMs

Synchronized with a clock signal

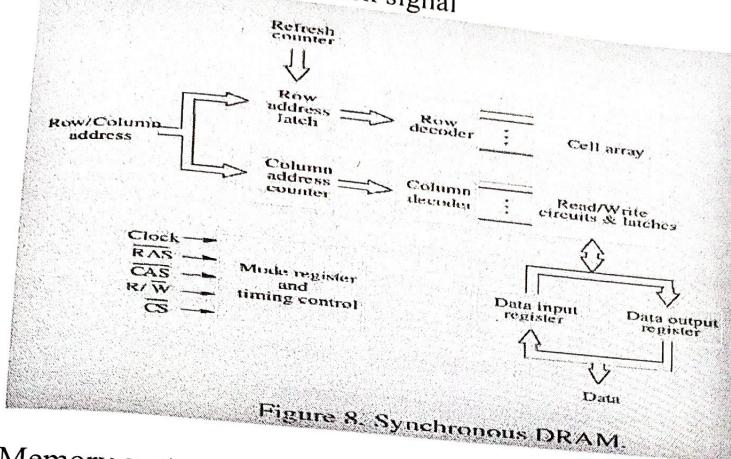
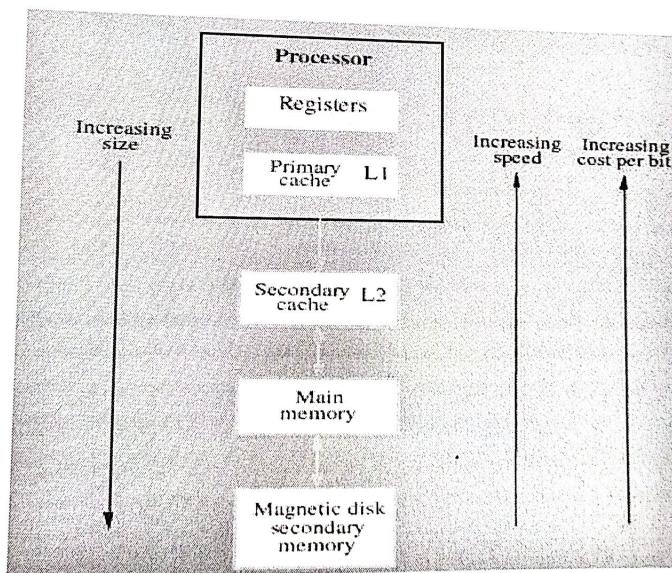


Figure 8. Synchronous DRAM.

Memory system considerations

- Cost
- Speed
- Power dissipation
- Size of chip



### Principle of locality:

**Temporal locality** (locality in time): If an item is referenced, it will tend to be referenced again soon.

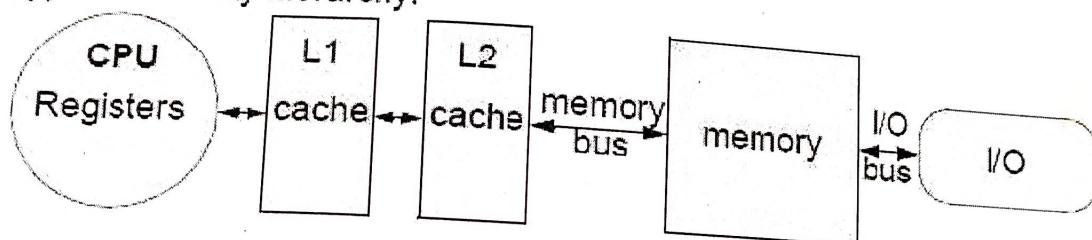
**Spatial locality** (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.

**Sequentiality** (subset of spatial locality).

The principle of locality can be exploited implementing the memory of computer as a *memory hierarchy*, taking advantage of all types of memories.

**Method:** The level closer to processor (the fastest) is a subset of any level further away, and all the data is stored at the lowest level (the slowest).

Typical memory hierarchy:



Level	1	2	3	4
Named as	Registers	Cache	memory	disk storage
Typical size	<1 KB	< 4 MB	<2 GB	>2GB
Access time (ns)	2 - 5	3 - 10	80 - 400	5'000'000
Bandwidth(MB/sec)	4000 - 32'000	800 - 5000	400 - 2000	4 - 32
Managed by	Compiler	Hardware	Operating system	Operating system / user

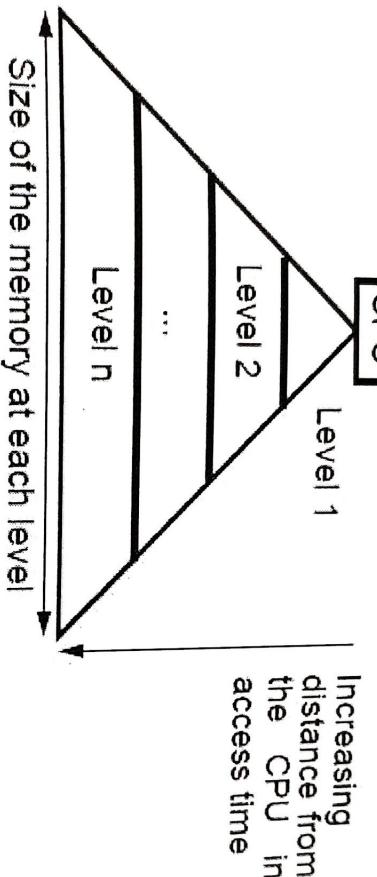
### Cache Memories

- Speed of the main memory is very low in comparison with the speed of processor
- For good performance, the processor cannot spend much time of its time waiting to access instructions and data in main memory.
- Important to device a scheme that reduces the time to ace the information
- An efficient solution is to use fast cache memory When a cache is full and a memory word

that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contain the referenced word.

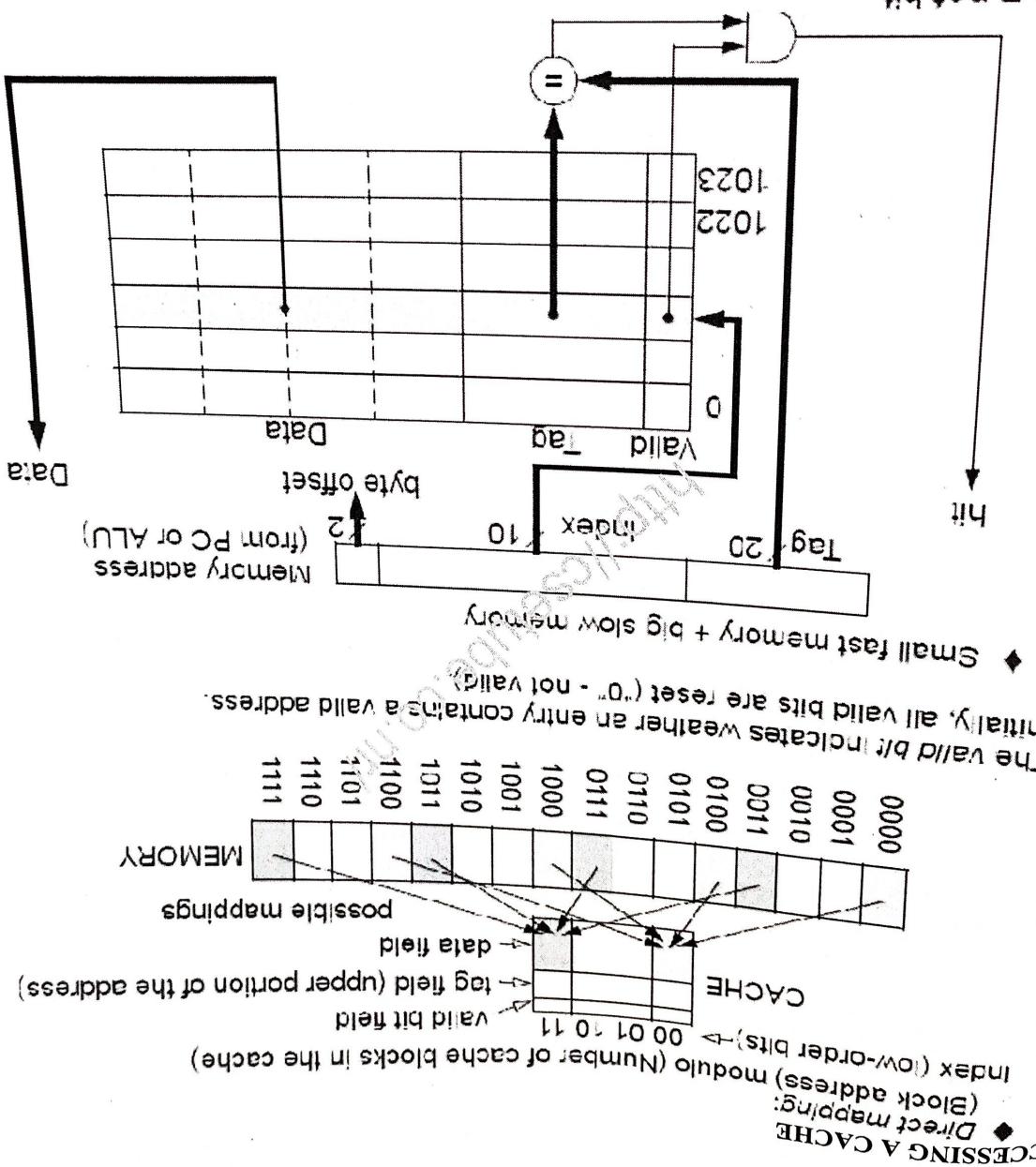
#### The basics of Caches

- " The caches are organized on basis of *blocks*, the smallest amount of data which can be copied between two adjacent levels at a time.
- " If data requested by the processor is present in some block in the upper level, it is called a *hit*.
- " If data is not found in the upper level, the request is called a *miss* and the data is retrieved from the lower level in the hierarchy.
- " The fraction of memory accesses found in the upper level is called a *hit ratio*.
- " The storage, which takes advantage of locality of accesses is called a *cache*





meaning the data has to be retrieved from lower level of memory system.  
miss = not hit.



### **virtual memory**

It is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow onto disk storage.

Virtual memory provides two primary functions:

1. Each process has its own address space, thereby not required to be relocated nor required to use relative addressing mode.
2. Each process sees one contiguous block of free memory upon launch. Fragmentation is hidden.

## **MEMORY MANAGEMENT REQUIREMENTS**

### Partitioning Strategies – Fixed

Fixed Partitions – divide memory into equal sized pieces (except for OS)

- Degree of multiprogramming = number of partitions
- Simple policy to implement
  - All processes must fit into partition space
  - Find any free partition and load the process

### Partitioning Strategies – Variable

Idea: remove “wasted” memory that is not needed in each partition  
Memory is dynamically divided into partitions based on process needs  
Definition:

- *Hole*: a block of free or available memory
- Holes are scattered throughout physical memory
- *External fragmentation*
  - memory that is in holes too small to be usable by any process
  - Allocate part (or all) of memory to process and mark remainder as free
- *Compaction*
  - Moving things around so that holes can be consolidated
  -

## **MEMORY MANAGEMENT POLICIES:**

- *First Fit*: scan free list and allocate first hole that is large enough
- *Next Fit*: start search from end of last allocation
- *Best Fit*: find smallest hole that is adequate slower  
and lots of fragmentation
- *Worst fit*: find largest hole

## **ASSOCIATIVE MEMORY**

- A content addressable processor is a content addressable memory with the added capability to write in parallel (multi-write) into all those words indicating agreement as the result of a search.

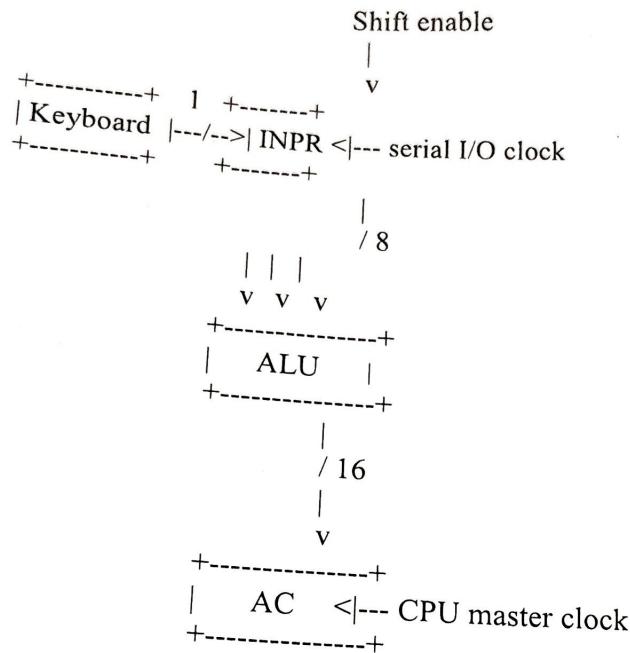
- A typical associative memory has the following components:

- Memory array
  - Comparand register
  - Mask register
  - Match/Mismatch (response) register
  - Multiple match resolver
  - Search logic
  - Input/Output register
  - Word select register
- 
- Mask Register is used to mask off portions of the data words which do not participate in the operations.
  - Word Select Register is used to mask off the memory words which do not participate in the operation.
  - Match/Mismatch Register indicates the success or failure of a search operation.
  - Input/Output Buffer acts as an interface between associative memory and the outside word.
  - Multiple Match Resolver narrows down the scope of the search to a specific location in the memory cell array in cases where more than one memory word will satisfy the search condition(s).
  - Some/None bit shows the overall search result.

## Input-Output and Interrupt

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor.

The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.



How many CPU clock cycles are needed to transfer a character from the keyboard to the INPR register? (tricky)

Are the clock pulses provided by the CPU master clock?

RS232, USB, Firewire are serial interfaces with their own clock independent of the CPU. ( USB speed is independent of processor speed. )

- RS232: 115,200 kbps (some faster)
- USB: 11 mbps
- USB2: 480 mbps
- FW400: 400 mbps
- FW800: 800 mbps
- USB3: 4.8 gbps

*OUTR* inputs are connected to the bus in parallel, and the output is connected serially to the terminal. *OUTR* is another shift register, and the printer/monitor receives an end-bit during each clock pulse.

### I/O Operations

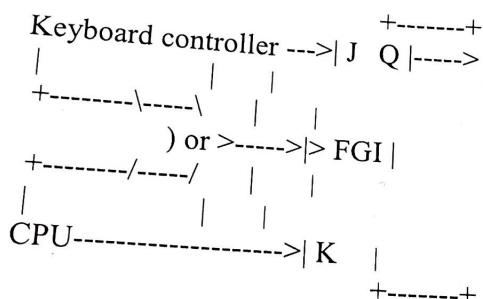
Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output.

The FGI flip-flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU).

The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a microoperation controlled by the CU, so we must include it in the CU design.

The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR.

Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop



How do we control the CK input on the FGI flip-flop? (Assume leading-edge triggering.)

There are two common methods for detecting when I/O devices are ready, namely *software polling* and *interrupts*. These two methods are discussed in the following sections.

Table 5-5 outlines the Basic Computer input-output instructions.

### Interrupts

To alleviate the problems of software polling, a hardware solution is needed.

Analogies to software polling in daily life tend to look rather silly. For example, imagine a teacher is analogous to a CPU, and the students are I/O devices. The students are working asynchronously, as the teacher walks around the room constantly asking each individual student "are you done yet?".

What would be a better approach?

With interrupts, the running program is not responsible for checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself!

When a device becomes ready, the CPU hardware initiates a branch to an I/O subprogram called an *interrupt service routine (ISR)*, which handles the I/O transaction with the device.

An interrupt can occur during *any* instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted.

Figure 5-13 outlines the Basic Computer interrupt process.

Interrupts can be globally enabled or disabled via the IEN flag (flip-flop).

Some architectures have a separate ISR for each device. The Basic Computer has a single ISR that services both the input and output devices.

If interrupts are enabled, then when either FGI or FGO gets set, the R flag also gets set. (R = FGI v FGO) This allows the system to easily check whether *any* I/O device needs service. Determining which one needs service can be done by the ISR.

If R = 0, the CPU goes through a normal instruction cycle. If R = 1, the CPU branches to the ISR to process an I/O transaction.

How much time does checking for interrupts add to the instruction cycle?

Interrupts are usually disabled while the ISR is running, since it is difficult to make an ISR reentrant. (Callable while it is already in progress, such as a recursive function.) Hence, IEN and R are cleared as part of the interrupt cycle. IEN should be re-enabled by the ISR when it is finished. (In many architectures this is done by a special return instruction to ensure that interrupts are not enabled before the return is actually executed.)

The Basic Computer interrupt cycle is shown in figure 5-13 (above).

The Basic Computer interrupt cycle in detail:

$T_0 T_1 T_2 (IEN)(FGI \vee FGO): R \leftarrow 1$