Syllabus

Module No.	Unit No.	Contents	No of Hrs.	CO
1		**Structure of a Computer System**		
	1.1	Introduction of computer system and its sub modules, Basic organization of computer and block level description of the functional units. Von Neumann model, difference between computer architecture and computer organization.	04	CO1
	1.2	Introduction to buses, bus types, and connection I/O devices to CPU and memory, PCI and SCSI		
2		**Arithmetic and Logic Unit**		
	2.1	Booth's Recoding and Booth's algorithm for signed multiplication, Restoring division and non-restoring division algorithms.	10	CO1
	2.2	IEEE floating point number representation and operations: Addition. Subtraction, Multiplication and Division. IEEE standards for Floating point representations :Single Precision and Double precision Format		
3		**Central Processing Unit**		
	3.1	CPU architecture, Register organization, Instruction formats and addressing modes(Intel processor).,Basic instruction cycle. Control unit Operation ,Micro operations : Fetch, Indirect, Interrupt , Execute cycle Control of the processor, Functioning of micro programmed control unit, Micro instruction Execution and Sequencing, Applications of Micro programming.	10	CO2
	3.2	RISC v/s CISC processors, RISC pipelining		
		Self learning: RISC and CISC Architecture, Case study on SPARC		
4		**Memory Organization**		
	4.1	Characteristics of memory system and hierarchy, Main memory, Cache memory principles , Elements of Cache Design.	10	CO3
	4.2	ROM, Types of ROM, RAM, SRAM, DRAM, Flash memory, High speed memories		
	4.3	Cache Memory Organization: Address mapping, Replacement Algorithms, Cache Coherence, MESI protocol, Interleaved and associative memories, Introduction to: Virtual memory, Main memory allocation, Segmentation ,Paging: demand paging and thrashing. Secondary storage, RAID levels		
5		**I/O Organization**		
	5.1	External Devices, I/ O Modules	03	CO4
	5.2	Programmed I/O, Interrupt driven I/O, DMA		

Module No.	Unit No.	Contents	No of Hrs.	CO
6		**Multiprocessor Configurations**		
	6.1	Flynn's classification, Parallel processing systems and concepts, Introduction to pipeline processing and pipeline hazards.	08	CO4
	6.2	Design issues of pipeline architecture, Instruction pipelining: Six Stage instruction pipeline.		
	6.3	8086 Instruction set (Arithmetic Instructions, Logical Instructions, Data transfer instructions), Assembly language programming.		
		Total	45	--

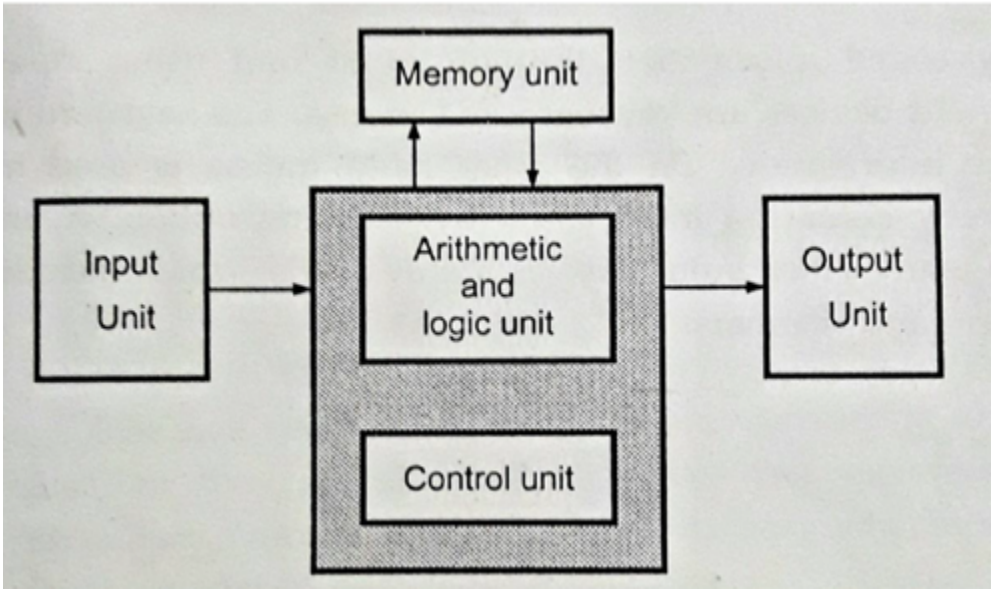
Module 1: Introduction- 5/10 marks

1.1 Architecture and Organization

Definition: Architecture is the attributes visible to the programmer
Example: Instruction set, number of bits used for data representation, I/O mechanisms, addressing techniques.

Definition: Organization is how features are implemented
Example: Control signals, interfaces, memory technology.

Basic Organization of Computer



1. Input unit: With input unit, data can be supplied to the computer from outside. Example: keyboard mouse.
2. Memory unit: The MU is used to store programs and data. Is of two types: primary and secondary storage memory devices.
   Primary storage is also called main memory/internal memory. It stores active data and programs. Volatile memory: RAM
   Secondary memory is called external/auxiliary memory. It is non volatile and used for permanent storage of data and programs. Non volatile memory: ROM and PROM
   Example hard disk, floppy disk.
3. ALU: Arithmetic and Logical unit is responsible for all the arithmetic operations like add subtract multiplication and division, as well as the logical operations of AND &&, OR ||, and inversion

To perform these operations, operands from the main memory are brought into high speed storage elements called **registers** of the processor.

4. Output unit: This sends the processed results to the user by means of output devices like the monitor, printer, etc.

5. Control unit: Responsible for the coordination and control of the activities among all functional units.
    - It fetches instructions stored in the main memory
    - Identifies which operations and devices are associated with it
    - And accordingly generates control signals to execute the desired operations.

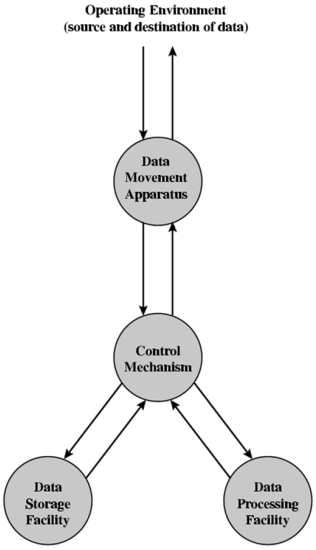
CU has a set of registers and a control circuit which generates the control signals. **The ALU along with CU is called CPU (Central processing unit)**

Difference between organization and architecture

COMPUTER ARCHITECTURE	COMPUTER ORGANIZATION
Way hardware components are connected together to form a computer system.	**Structure and behaviour** of a computer system as **seen by the user**.
It acts as the interface between hardware and software.	It deals with the components of a connection in a system.
Helps us to understand the **functionalities** of a system.	How exactly all the **units in the system are arranged and interconnected.**
A programmer can **view architecture** in terms of **instructions, addressing modes and registers**.	Whereas Organization expresses the **realization of architecture.**
While **designing** a computer system **architecture is considered first**.	An organization is done on the basis of architecture.
Computer Architecture deals with high-level design issues.	Computer Organization deals with low-level design issues.
Architecture involves **Logic** (Instruction sets, Addressing modes, Data types, Cache optimization)	Organization involves **Physical Components** (Circuit design, Adders, Signals, Peripherals)

Aspect	Computer Architecture	Computer Organization
Definition	Way hardware components are connected to form a computer system.	Structure and behavior of a computer system as seen by the user.
Focus	Acts as the interface between hardware and software.	Deals with the components of a connection in a system.
Purpose	Explains the **functionalities** of a system.	Explains how all the **units are arranged and interconnected**.
Viewpoint	Programmer sees architecture in terms of **instructions, addressing modes, and registers**.	Expresses the **realization of architecture**.
Design Order	Considered **first** when designing a system.	Organization is done **based on architecture**.
Design Level	Concerned with **high-level design issues**.	Concerned with **low-level design issues**.
Involves	**Logic**: Instruction sets, addressing modes, data types, cache optimization.	**Physical Components**: Circuit design, adders, signals, peripherals.

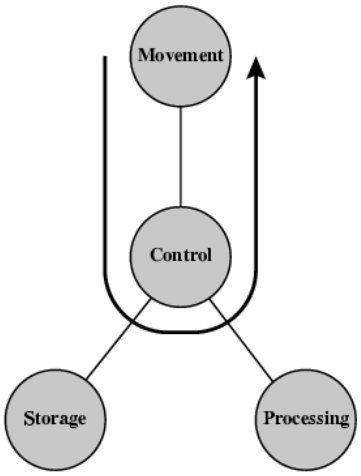
Structure is the way in which different components relate to each other. It tells how components are arranged and how they connect.
Function is an operation of individual components. Tells what each component does.



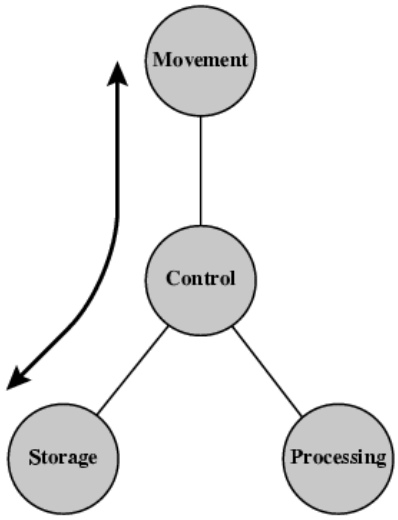
Function

Computer functions include:

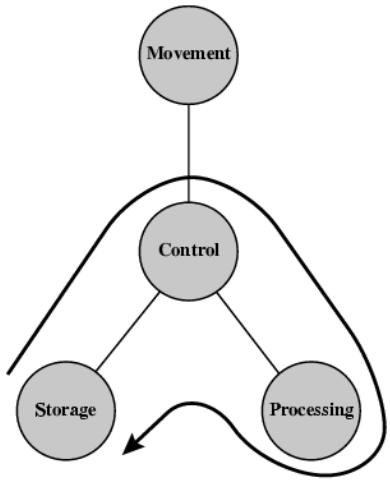
1. **Data processing**
2. **Data storage**: Files are stored on the computer for retrieval and update.
3. **Data movement**: Computers act as sources and destinations of data. When data is retrieved or delivered to or from a computer, the process is called IO (input-output), and the device is called a **peripheral.**
   When data is moved across a longer distance or from a remote device, the process is called **data communications.**
4. **Control:** Manages the resources of the computer. The control unit directs all components to work together as per instructions.



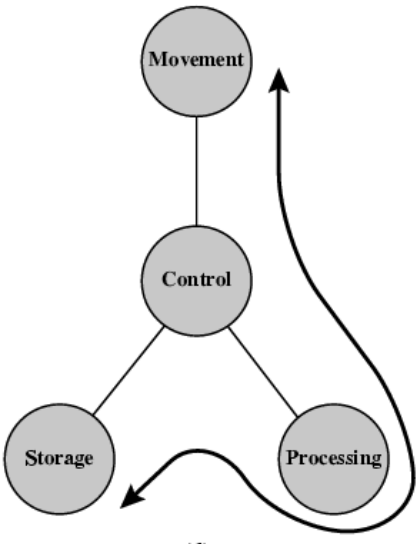
Data movement: Transferring data from one location to another.



Data storage: input->control->storage (read)
Storage ->control->output (write)



Processing from/to storage: storage se data is picked->processes->sends data back to storage



Processing from storage to IO

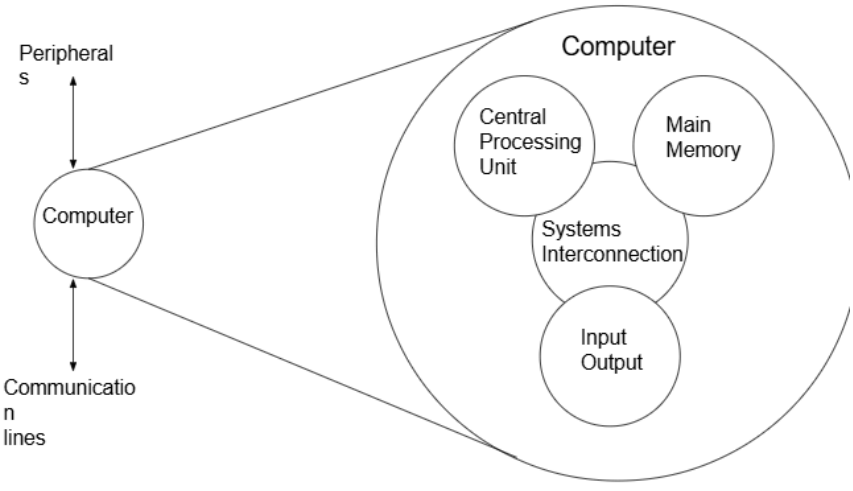
Fetch data from storage -> process -> send to output

Structure: Computer

Computer has many parts, CPU is one of them.

Top down

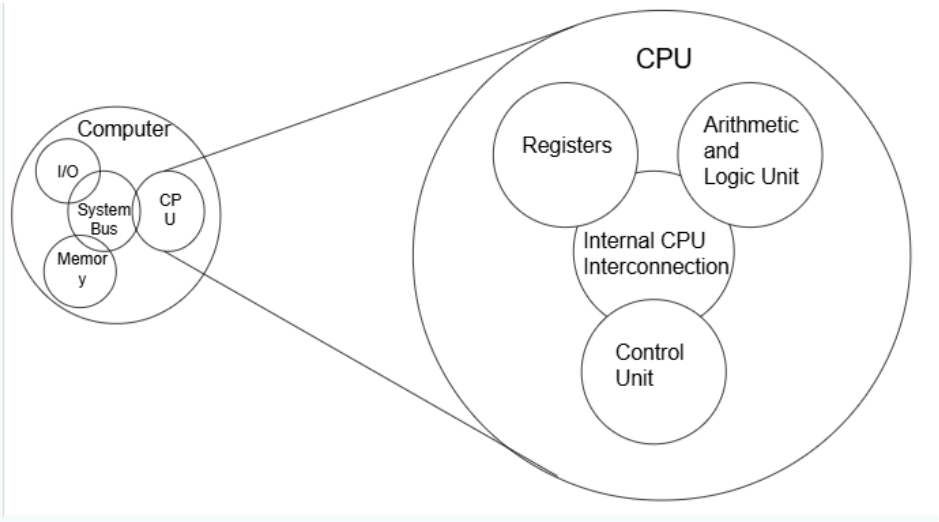
Meaning: begins with a top view and decomposes the system into its subparts successively.



- **CPU**: controls the operations of the computer and performs its data processing.
- **Main memory**: stores data and instructions

- **Io:** moves data between the computer and the external environment (hardware like monitor keyboard mouse)
- **System interconnection:** A mechanism that helps communication between CPU, memory, and IO. example: **System bus** that consists of **many conducting wires.** Other components are attached to these wires.

Structure: CPU

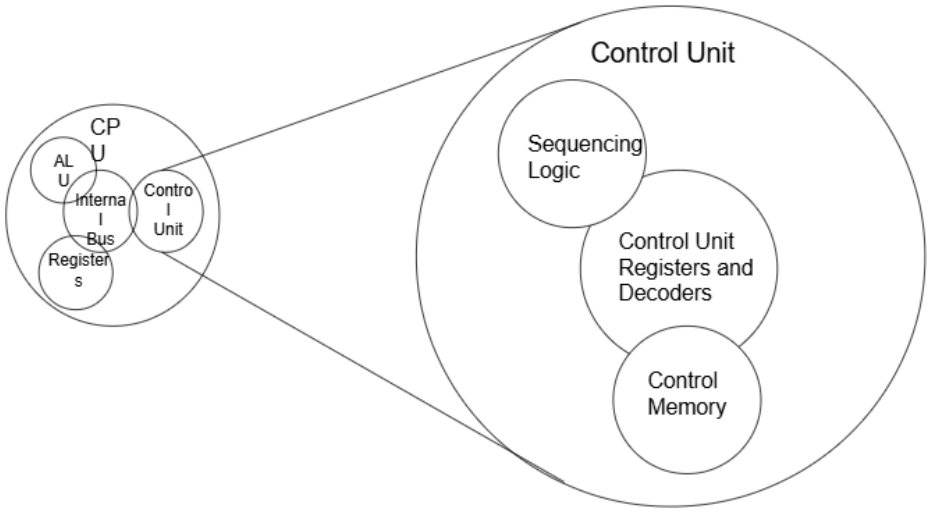


- **Control unit:** Controls the operations of the cpu
- **ALU:** concerned with the arithmetic and logical operations
- **Registers:** Provide internal storage to the CPU.
- **CPU interconnection:** Mechanism for communication between control unit, registers, and ALU.

A microprogrammed control unit runs microinstructions stored in control memory to generate control signals.
Other type: hardwired control unit.

Structure: Control unit

Control unit ke parts



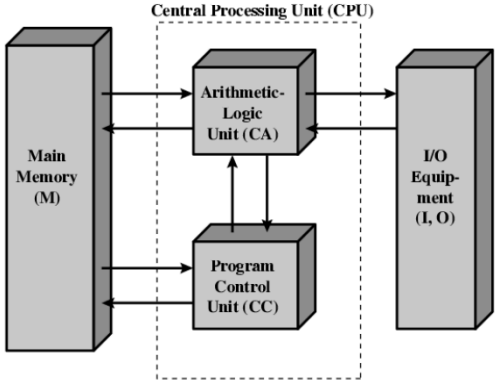
Multicore computer structure

When all processors reside on a single chip, we call it a multicore computer

1. **Cpu**: the portion of a computer that fetches and processes instructions. Comprises of alu, cu, and registers. Single cpu= one or more cores. Each core = independent small cpu.
2. **Core**: a single processing unit on a single processor chip. Core = cpu in functionality on a single cpu system.
3. **Processor**: The computer component (physical chip) responsible for interpretation and execution of instructions. If a processor has more than one core, it's called a multicore processor.
   Processor = physical chip.
   CPU = logical unit on that chip.

Von Neumann

Structure of Von Neumann machine



Fixed program computers

Computers like calculators have a very specific and un-reprogrammable function.

Stored program computers

Programs and data are stored in a separate storage unit called “memories”. They’re treated similarly and make it easier to reprogram.

It uses a single processor, a single memory for both instructions and data, and executes programs by the **fetch-decode-execute** cycle.

Definition of von neumann architecture: A computer design where program and data share the same memory, and execution follows the cycle: fetch → decode → execute.

- Main memory stores programs and data.
- ALU operates on binary data.
- CU interprets instructions from the memory and executes them.
- Input and output are also operated by the CU. I/O devices worked only when the CU issued commands.

Also called IAS (Institute for Advanced Study computer)

Important Diagram for IAS

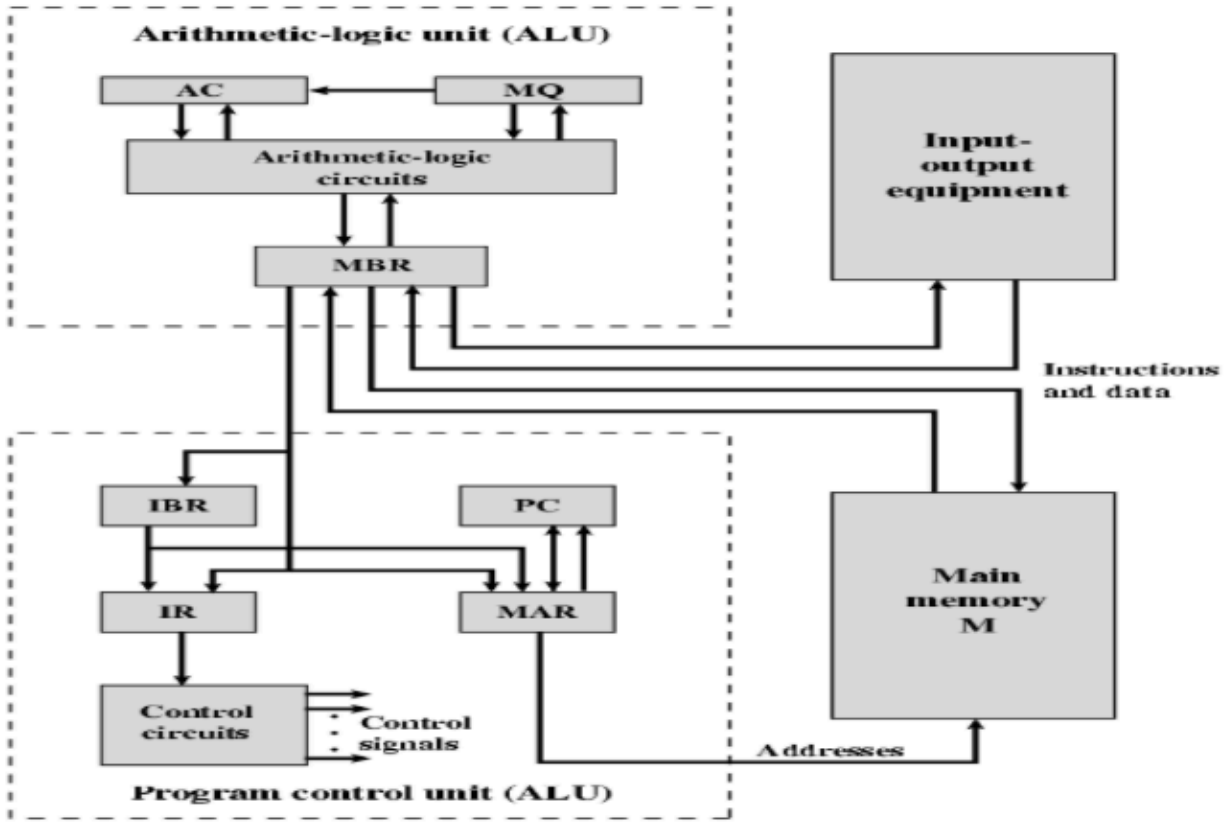


Diagram:

1. IBR connects to MAR and IR
2. MBR connects to IBR IR AND MAR singly
3. MBR connects to main memory and io doubly
4. Ir connects to cc
5. Mar connects to pc and main.

6. MQ and AC doubly connect to AL circuits
7. ALC doubly to MBR
8. Mq to AC

- **AC**: Central arithmetic part which is responsible for elementary arithmetic and logical operations.
- **CC (Program control unit)**: Central control: Responsible for proper sequencing of operations and logical control of the device.
- **M:** Collection of storage cells and circuits. Responsible for transferring memory in and out of storage. Uses binary information called “words”.
- **Io:** Capable of delivering and retrieving data as output and input.
- Both the ALU and CU contain registers (registers are storage locations)

Registers[7]

1. **Memory buffer register MBR**
   Contains a word(group of bits) to be stored in memory / be sent to the IO unit.
   Also used to receive a word from memory or IO.
   MBR holds data being transferred to or from memory or I/O.
2. **Memory address register**
   Memory Address Register (MAR) holds the memory location address for the data in MBR.
3. **Instruction register**
   Instruction Register (IR) holds the opcode of the current instruction being executed.
   Opcode: The part of an instruction that tells the CPU which operation to perform (e.g., add, load, store).
4. **Instruction buffer register**
   Acts as a buffer and stores the next instruction in-line while the previous is being run.
   a. Helps improve performance of instruction execution.
5. **Program counter**
   Contains the address of the next instruction to be fetched from memory.
6. **Accumulator and multiplier quotient**
   AC & MQ registers temporarily store operands and results of ALU operations.

1. 2 Bus

What is a bus?

- A communication pathway connecting two or more devices
- Connects CPU, memory, and I/O devices
- Made of parallel lines
- A set of lines that transmits data, addresses, or control signals. Can send multiple bits simultaneously (e.g., 32-bit data bus).
- Shared transmission medium: multiple devices are connected to the bus. A signal transmitted by one device = available for reception by all other devices connected to this bus.
- Two devices transmit signals at the same time = signals overlap and become garbled(distorted). At a time, only one device can successfully transmit.
- Consists of multiple communication pathways.
    - **System bus:** A bus that connects major computer components like memory processor, IO.
    - **Northbridge**: links the CPU to very high speed devices, like RAM and graphic controllers.
    - **Southbridge**: links the CPU to lower speed peripheral buses such as PCI or ISA
- Many different possibilities of interconnection systems in buses. Single and multiple BUS structures are most common: PC and unibus.

Functions:

1. **Data sharing:** Allows components to share data with computers' CPU and memory. By serial or parallel buses. 8 to 64 bit buses.
2. **Addressing**: A bus has address lines, which allows it to send data to or from specific memory locations. Address lines specify memory and IO locations.
3. **Power**: some buses can supply power to various devices
4. **Timing**: The system clock generates regular pulses that act as heartbeats. They synchronize the CPU, memory, and peripheral devices. This ensures that all parts work in step with each other.

Example:
An expansion bus helps connect components on the computer like a TV card or a sound card.

Bus structure

A system bus consists of about 50-100 lines (wires). Lines are classified into data, address, and control lines.

Data bus = Data lines

- Transfers actual data/instructions between CPU, memory, I/O.
- Bidirectional → allows read and write.
- Width (e.g., 8, 16, 32, 64 bits) decides how much data moves at once.
- Wider bus = higher data transfer speed.

Address bus

- Carries memory or I/O addresses.
- **Unidirectional** (CPU → memory/I/O).
- Width decides maximum addressable memory.
- Example: 32-bit address bus → 2^{32} locations.
- Memory capacity = $2^{\text{number of address lines}}$

//if 8080 has a 16 bit address lines. Its memory capacity = 2^{16} bytes
//should know size conversions. 1gb=?mb and so on

Control bus

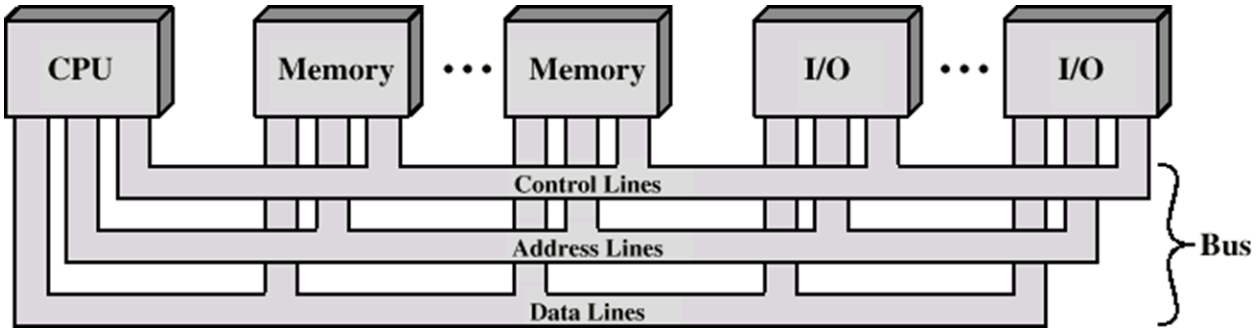
- Controls access to data and address lines
- Control usage of data and address lines
- Transmits the command (what to perform) and timing information(how long this data and address is valid)
- Carries control signals generated by CU
- Coordinates actions of CPU memory and IO
- Bidirectional

Examples of control lines

- **Memory write:** Causes data on the bus to be written into the addressed location
- **Memory read:** Causes data from the addressed location to be placed on the bus
- **I/O write:** Causes data on the bus to be output to the addressed I/O port
- **I/O read:** Causes data from the addressed I/O port to be placed on the bus
- **Transfer ACK:** Indicates that data have been accepted from or placed on the bus
- **Bus request:** Indicates that a module needs to gain control of the bus
- **Bus grant:** Indicates that a requesting module has been granted control of the bus
- **Interrupt request:** Indicates that an interrupt is pending
- **Interrupt ACK:** Acknowledges that the pending interrupt has been recognized
- **Clock:** Is used to synchronize operations
- **Reset:** Initializes all modules

Bus interconnection scheme

//important diagram



If a module wants to send data to another module,
   1. Obtains the use of a bus
   2. Transfers data via the bus

If a module wants to request data from another
   1. Obtain use of a bus
   2. Transfer a request to the other module over the Control and Address lines. Then, wait for the module to send data.

Module A **obtains bus access**.
Module A **sends request** (via control & address lines) to Module B.
Module A **waits**.
Module B **obtains bus access** and **transfers the requested data** back to Module A.

Problems with Single Bus

A lot of devices working with a single bus can have problems.

1. **Propagation delays:** Long data paths = bad coordination of bus can affect performance. Use multiple buses to overcome this problem. (Leads to contention, this implies that if we use more buses, it's more likely that two devices will try using the same bus at the same time = contention)
2. **Capacity:** If data transfer approaches bus capacity, the bus starts functioning slower.

The solution is to increase data rate by using a **wider bus**.
Best solution = use multiple buses
Can also use physical dedication (Physical dedication → give a private bus to a device → no sharing → no contention)

Bus types

Dedicated

- Have been permanently assigned to one function or to a physical subset of components.
- Have separate lines for control, data, and address bus. (each bus has its own line)
- Faster since no sharing of line takes place.
- More hardware = costlier and less flexible

Multiplexed

- Have shared lines, address valid or data valid control lines.
  Address valid line → signals that the current value on the bus is a valid memory or I/O address.
  Data valid line → signals that the current value on the bus is actual data.
- The advantage = fewer lines. Less hardware = cheap.
- The disadvantage = a more complex control and slower speed as functions share the same line.

Physical dedication

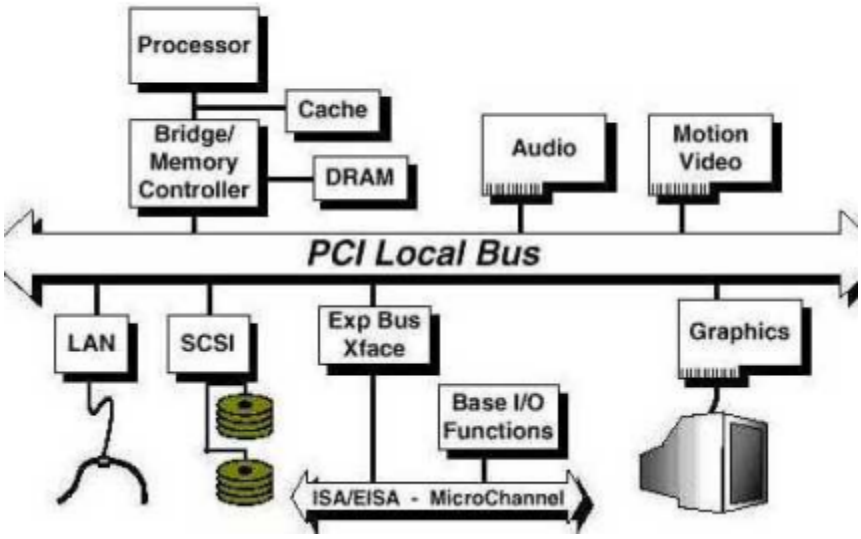
Use of multiple buses. Each connects only a subset of modules.

1. Example: One I/O bus connects all I/O modules
2. Each module type (CPU, memory, I/O) gets its own bus line.
3. Only that module type uses that line → bus contention disappears.
4. Increased size and cost
5. Middle ground between dedicated and multiplexed.

Bus arbitration

- At one time, only one module can control the bus. If more than one try to control, we increase the risk of data corruption and instability in the system.
- **Definition:** Bus arbitration is the process to decide which device gets bus control.
- Ensures orderly and fair access
- Arbitration may be **centralized** (a single controller decides who gets access. Simple and fast but bottleneck controller, so it slows the whole system) or **distributed** (devices negotiate among themselves. More reliable but complex. Removes bottleneck.)

PCI: Peripheral Component Interconnect



- PCI is an **expansion bus**.
- Parallel
- The PCI bus **connects the CPU and expansion boards**
  Examples: Modem, network card, sound and video card

- Follows a **high speed bus standard**: a set of rules for a bus that allow fast, reliable data transfer between components.
- **Plug and play**: auto detects the devices when connected
- 32 or 64 bit combination: parallel communication
- Supports **multiple devices** sharing the bus.
- **Combination** of address bus, data bus, and control bus
- The **PCI bridge** links CPU bus with PCI bus. This allows smooth communication between processor and peripherals.
- **High bandwidth, processor independent, and low cost.** It can function as a peripheral bus (bus used to connect external devices).
- PCI supports many microprocessor setups, including single- and multi-processor systems, using synchronous timing and centralized bus arbitration.

Features

1. **Plug & Play** → auto-detects devices.
2. **Hot-plug / Hot-swappable** → add/remove devices without shutting down.
3. **High speed** → up to 133 MHz.
4. **Backward compatible** → older independent bus devices can connect to PCI.
5. **Independent bus** → devices connected to PCI can operate without CPU intervention.
6. Supports 3 address spaces:
    a. Memory
    b. I/O (for processor use)
    c. Configuration (for PnP setup)

PCI bus lines //can skip

1. **System lines:** includes clock (synchronization) and reset (re-initialization)
2. **Address and data**
    a. Interrupt (A device sends signal to bus asking for attention) and validate (tells if current data/address on the bus is valid) lines
    b. Multiplexed (line is shared among both address and data buses)
3. **Interface control**: control the timing of transactions. Provides coordination.
4. **Arbitration**: Decides which device can use the bus at what time
5. **Error lines** : signal problems during transactions
6. **Interrupt lines**: allow devices to signal the CPU asynchronously. When a device needs CPu attention, it asserts its interrupt line.

7. **Cache support**: indicates that a read data transaction can be cached by the CPU.
8. **64-bit Bus Extension:** Adds 32 extra time-multiplexed lines for addresses and data. These combine with the main address/data lines to form a 64-bit bus. It helps interpret and validate the address and data signals.

PCIe (PCI express) is a standard for connecting devices to computers. It's software-compatible with PCI but has higher potential bandwidth and greater flexibility than traditional PCI.

Example of PCI implementation behind the scenes

//prolly just for understanding

1. You install the PCI sound card into the motherboard and power up.
2. The PnP BIOS scans the PCI bus, detects the new card, and assigns IRQ, DMA, memory, and I/O settings, saving them in the ESCD.
3. Windows XP boots, detects the new hardware, installs drivers automatically or via the wizard.
4. Once ready, audio from an external device is fed into the sound card.
5. The sound card converts analog to digital and sends data via the PCI bus.
6. The bus controller prioritizes the data, sending it to the CPU or system memory via the system bus.
7. The recorded data is stored in memory or saved to the hard drive for processing.

Illustrates how the PCI bus allows automatic detection, configuration, and transfer of data.

SCSI: Small computer system interface

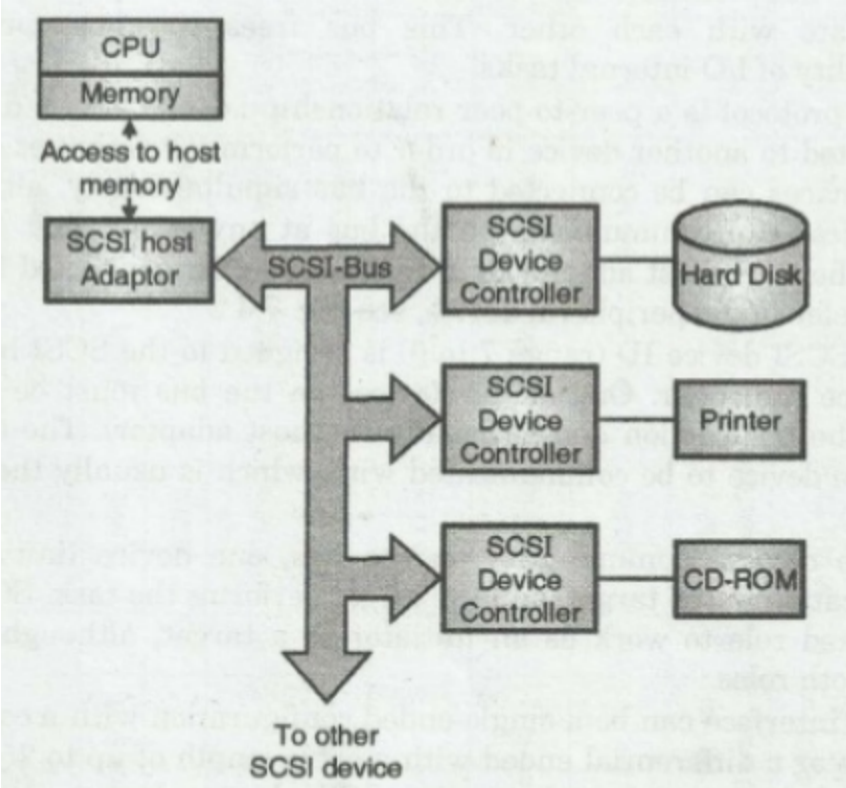


Fig. 7.4.2 : SCSI bus with host adapter and device controller

- SCSI is a set of standard electronic interfaces that allow computers to communicate with peripheral hardware.
- Each SCSI has a unique id
- Can connect many devices at the same time. Devices include hard drives, scanners, cds, printers, etc. helps put many items on one bus.
- Faster
- More flexible
- SCSI standards are generally backward compatible. This means a newer SCSI can work with older SCSI devices.
- SCSI interfaces have been replaced, for the most part, by Universal Serial Bus (USB) [but a usb can only connect small devices]
- Can add up to 15 devices
- Widely used in workstations, servers, and mainframes.
- Originally, parallel bus type. Later became series attached scsi.
- Plug and play: auto detect if a peripheral is connected
- **Disadvantages**:
    - More expensive than its alternatives
    - Complex: requires IDs, terminators
    - Has to be configured for each computer. All different SCSI types have different speeds/ widths/ connectors. Hard to keep track.

    - USB, SATA, and NVMe are cheaper, easier, and faster for most devices.
- Internal devices connect with a ribbon cable.
- If a SCSI bus is left open, electrical signals sent down the bus can reflect and interfere with the normal workings of the bus. So it is best to terminate the bus and close each end with a resistor circuit. If the bus supports both internal and external devices, the last device on each series should be terminated.
- **Advantages:**
    - Can be used for a wide range of peripheral devices.
    - Many devices accommodate on a single bus
    - Intelligent interface
    - Industry longevity : has been used for a long time in the industries, so more reliable
    - Termination: SCSI standards automatically configure drive settings

End of theory

Module 2:ALU-10 marks

Theory:

Booth's recoding and booth's algo difference

Booth's Algorithm	Booth's Recoding
1. Complete **multiplication method**.	1. **Preprocessing** technique only.
2. Uses **bit pairs (00,01,10,11)**.	2. Converts number to **{-1,0,+1}** digits.
3. Handles **runs of 1s** efficiently.	3. Reduces number of **non-zero digits**.
4. Performs **add/sub + shift** steps.	4. Only rewrites the multiplier; **no shifting** here.
5. Used directly by hardware to multiply.	5. Used **before** multiplication to simplify work.

How is booth's recoding better than booth's algo? -ise

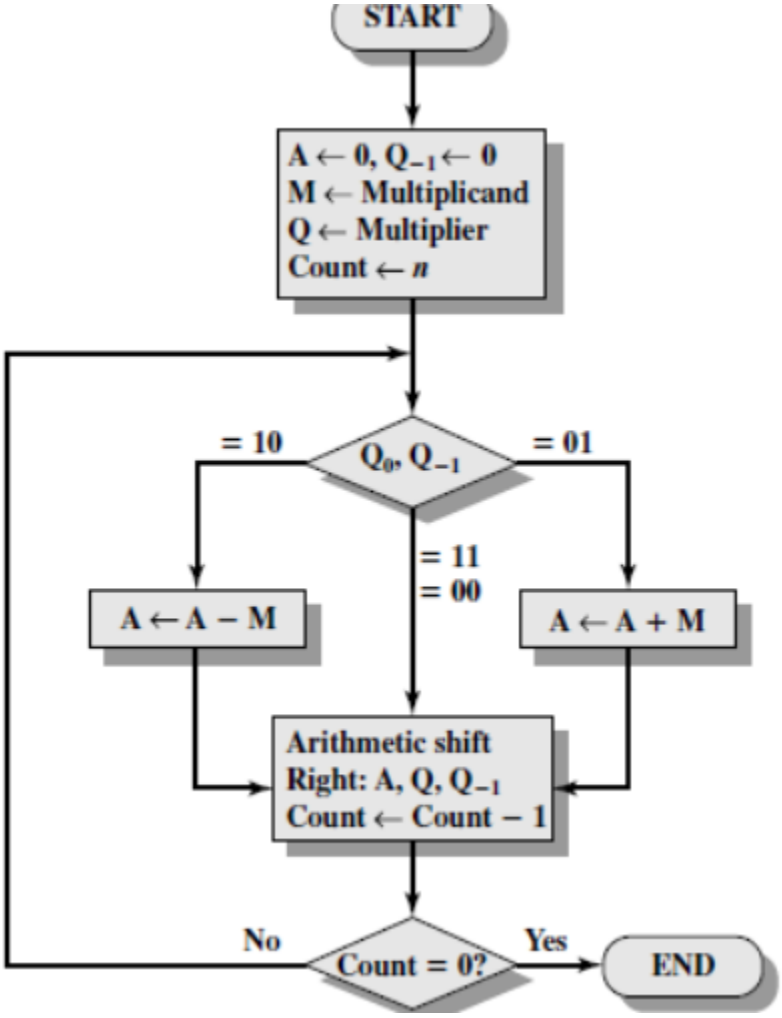
- Booth's recoding reduces the number of non-zero digits. This means fewer add/sub operations during multiplication. It makes the multiplication process faster.
- It converts bits into −1, 0, +1 form for simplicity. This form avoids long runs of 1s more efficiently than Booth's algorithm.
- Recoding creates a cleaner, shorter sequence of operations.
- It reduces switching activity inside hardware. This lowers power consumption. It also lowers the chances of carry-propagation delay.
- Recoding improves performance for large multipliers.
- It simplifies hardware design for signed multiplication.

Overall, Booth's recoding is an optimized version that speeds up Booth's algorithm.

What is the difference between restoring and non restoring division-2M

Restoring Division	Non-Restoring Division
1. After a wrong subtraction, remainder is **restored**.	1. No restoring step is used.
2. More steps due to extra restore.	2. Faster because restore is skipped.
3. Algorithm is simpler.	3. Algorithm is slightly complex.
4. More hardware time needed.	4. Less hardware time needed.
5. Remainder always maintained positive.	5. Remainder can go negative temporarily.

Booth's Algorithm



10 = A-M
01 = A+M
M is the multiplicand and Q is the multiplier.

If 20 = 10100 in binary, write -20 as:
010100 <extra 0 for sign)
101011 <take complement>
+1
= 101100<binary addition>

Method:

a. Make four columns. A, Q, Q-1, and N
b. A starts with x number of off bits - 0000 for x=4
c. Q is the multiplier

d. Q-1 starts with 0
e. N starts with x, the number of bits
f. At each iteration, shift bits to the right.
g. If you get unequal bits in q-1 and q's last bit, do operation as mentioned in booth's algo: 01 hai toh a=a+m replace and if it's 10, replace a with a-m.
h. Continue iterating with updated values of a whenever updated, whenever you shift bits to the right, decrease counter.
i. When counter hits 0, stop process. Merge a and q to get the final answer.
j. If one number is negative and other is positive, the final answer needs to be 2's complemented before submitting. {example 1}
k. If both numbers are positive or negative, the answer is positive. Leave answer as it is after converting to decimal. {example2}

1. -10 * 10
   // m, a and q need to have equal bits.
   //where m*q=a
   Minimum 4 bits are needed.
   Q = 01010 <binary for 10>
   M = 10101
   +1
   M= 10110 <2's compliment>

a	q	q-1	n	verdict
00000	0101 0	0	5	Right shift
00000	0010 1	0	4	1 0 in q and q-1, do a-m
01010 =a-m	0010 1	0		Right shift
00101	0001 0	1	3	0 1, do a+m
A+m = 11011	0001 0	1		Right shift
11101	1000 1	0	2	10, do a-m
A-m = 00111	1000 1	0	2	Right shift
00011	1100 0	1	1	01, a+m
11001	1100 0	1	1	Right shift
11100	1110 0	0	0	counter=0, stop iteration

1110011100

0001100011
+1
1100100 = 64+32+4 = -100 answer

2. 11 * 3
// m, a and q need to have equal bits.
//where m*q=a
Q = 01011 <binary for 11>
M = 00011
11100
+0001
11101
M= 11101<2's compliment>

a	q	q-1	n	verdict
00000	01011	0	5	10, a-m
11101	01011	0	5	Right shift
11110	10101	1	4	Right shift
11111	01010	1	3	01, a+m
00010	01010	1	3	Right shit
00001	00101	0	2	10, a-m
11110	00101	0	2	Right shift
11111	00010	1	1	01, a+m
00010	00010	1	1	rs
00001	00001	0	0	stop

Ans: 0000100001 = 33

3. -8*-8

M = -8
8 = 01000
-8 -> 10111
+1
M = Q = 11000

a	q	q-1	n	verdict
00000	11000	0	5	Right shift

00000	01100	0	4	
00000	00110	0	3	
00000	00011	0	2	10, do a=a-m
01000	00011	0	2	rs
00100	00001	1	1	
00010	00000	1	0	Counter 0 stop

Ans: 0001000000 = 64

4. 15*-10

M = 15 = 01111
Q = -10 = 10110 <derived in ex 1.

a	q	q-1	n	verdict
00000	10110	0	5	Right shift
00000	01011	0	4	10, a-m
01111	01011	0		rs
00111	10101	1	3	11,rs
00011	11010	1	2	01, a+m
10100	11010	1		rs
11010	01101	0	1	10, a-m
01001	01101	0	1	
00100	10110	1	0	stop

Ans; 10010110 ka complement = 101101010

5. m=19 and q=-20
   M = 10011
   Q = 10100
   M = -19 = 101101
   Q = -20 = 101100

a	q	q-1	n	verdict
000000	101100	0	6	
000000	010110	0	5	
000000	001011	0	4	10, a-m
010011	001011	0		
001001	100101	1	3	
000100	110010	1	2	01, a+m
110001	110010	1		
111000	111001	0	1	10, a-m
001011	111001	0		
000101	111100	1	0	STOP

Ans; A+Q concatenation'
101111100 = 380

6. M=15, Q=-6
M = 01111
-M = 10001
-Q = 00110
Q = 11010

a	q	q-1	n	verdict
00000	11010	0	5	
00000	01101	0	4	10, a-m
10001	01101	0		ok
11000	10110	1	3	01, a+m
00111	10110	1		
00011	11011	0	2	10,a-m
10100	11011	0		
11010	01101	1	1	
11101	00110	1	0	stop

Ans: 1110100110 's two's compliment

Ans = 1011010 <using 2's compliment of 1110100110>
Ans = -90

7. M= 48 -> 0110000
   Q = -3 -> 1111101
   -Q = 3 = 0000011
   -M = 1010000c

a	q	q-1	n	verdict
0000000	1111101	0	7	10, a-m
1010000	1111101	0		
1101000	0111110	1	6	01, a+m
0011000	0111110	1		
0001100	0011111	0	5	10, a-m
1011100	0011111	0		
1101110	0001111	1	4	
1110111	0000111	1	3	
1111011	1000011	1	2	
1111101	1100001	1	1	
1111110	1110000	1	0	stop

   11111101110000 compliment = 10010000 = 144

Practice

1. -29*12
   M = 001100
   -M = 110100
   Q = 100011

a	q	q-1	n	verdict
000000	100011	0	6	10->a-m
110100	100011	0		rs
111010	010001	1	5	rs

111101	001000	1	4	01->a+m
001001	001000	1		rs
000100	100100	0	3	rs
000010	010010	0	2	rs
000001	001001	0	1	10->a-m
110101	001001	0		rs
111010	100100	1	0	stop

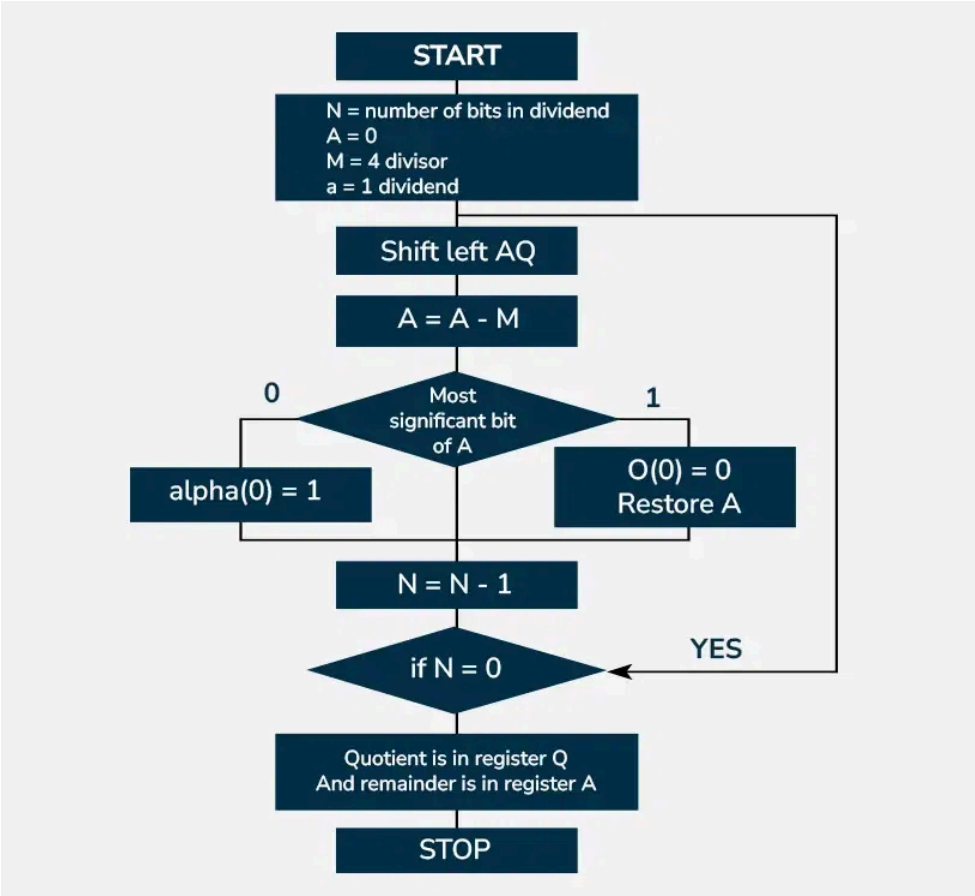
2. -16 * -23
   16 = 010000 = -m
   -16=110000 = m
   23 = 010111
   -23 = 101001 = q

a	q	q-1	n	verdict
000000	101001	0	6	10-> a-m
010000	101001	0		rs
001000	010100	1	5	01->a+m
111000	010100	1		rs
111100	001010	0	4	rs
111110	000101	0	3	10->a-m
001110	000101	0		rs
000111	000010	1	2	01->a+m
110111	000010	1		rs
111011	100001	0	1	10->a-m
001011	100001	0	1	rs
000101	110000	1	0	stop

   -16 * -23 = 368
   368 = 000101110000

Restoring Division

flowchart



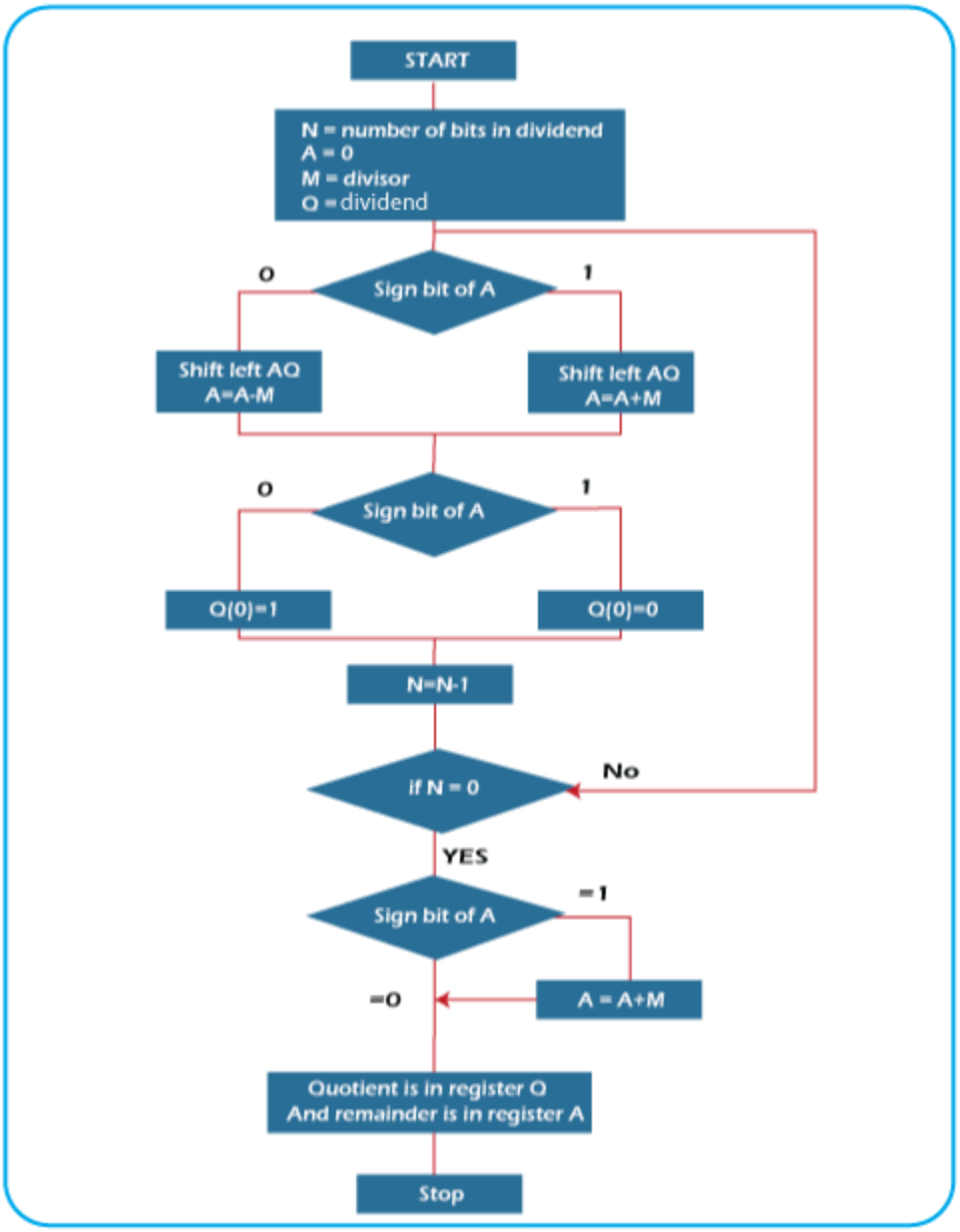
Solution:

9) 1/3

n	A	q	
4	0000	0001	SL
	0000	001 –	A – = M
	1101	0010	<0 ? yes
	0000	0010	A = A + M
3	0000	010 –	SL
	1101	010 –	A = A - M
		0100	A < 0 ? yes.
	0000	0100	A = A + M
2	0000	100 –	SL
	1101	100 0	A = A - M
	0000	1000	A < 0 ? yes.
1	0000	1000	A = A + M
	~~0001~~	~~000 1~~	A = A - M
	0001	000 –	
	1110	0000	A < 0 ? yes
0	0001 Rem	0000 Q	A = A + M

Non Restoring Division

Algorithm



Difference between restoring and non-restoring:

1. We are not changing A for each step, we are just checking if A has gone to negative. In restoring, we were restoring A's value after each iteration.

Non Restoring Div Questions:

1. 7/3

M = 3 = 0011
Q = 7 = 0111
-M = 2's complement of 3
1100
+ 1
1101 = -M

A	Q	n	Verdict	Task
0000	0111	4	A<0? no.	Shift left; a=a-m
0000	111_	4		
1101	111_	4	A<0? yes	Bit = 0
	1110			
1101	1110	3	A<0? yes	Shift left, a+m
1011	110_	3		
1110	110_	3	a<0? yes	bit=0
1110	1100	2	a<0? Yes.	Shift left, a+m
1101	100_	2		
0000	100_	2	a<0? no	bit=1
0000	1001	1	a<0? no	Shift left, a-m
0001	001_	1		
1110	001_	1	a<0? yes	bit=0
1110	0010	0		stop
			a<0; a=a+m	
0001	0010	0		

Remainder= A = 1 = 0001
Quotient= Q = 2 = 0010
Correct.

2. 20/7

M = 7 = 000111
Q = 20 = 010100

-M = 2’s complement of 7
111000
+ 1
111001 = -M

A	Q	n	Verdict	Task
000000	010100	6	a<0? no	Shift left, a-m
000000	10100_	6		
111001	10100_	6	a<0? yes	bit=0
111001	101000	5	a<0? yes	Shift left, a+m
110011	01000_	5		
111010	01000_	5	a<0? yes	Bit =0
111010	010000	4	a<0? yes.	Shift left, a+m
110100	10000_	4		
111011	10000_	4	a<0? yes	Bit =0
111011	100000	3	a<0? yes	Shift left, a+m
110111	00000_	3		
111110	00000_	3	a<0? yes	Bit =0
111110	000000	2	a<0? yes	Shift left a+m
111100	00000_	2		
000011	00000_	2	a<0? no	bit=1
000011	000001	1	a<0? no	Shift left, a-m
000110	00001_	1		
1111111	00001_	1	a<0? yes	Bit =0
1111111	000010	0		stop
			a<0?	a=a+m
000110	000010			

Remainder= A = 6 = 000110
Quotient= Q = 2 = 000010

Correct.

3. 29/10

M = 10 = 001010
Q = 29 = 011101
-M = 2's complement of 10
110101
+ 1
110110 = -M

A	Q	n	Verdict	Task
000000	011101	6	a<0? no	Left shift, a-m
000000	11101_	6		
110110	11101_	6	a<0? yes.	bit=0
110110	111010	5	a<0? yes	Left shift
101101	11010_	5		a+m
110111	11010_	5	a<0? yes	bit=0
110111	110100	4	a<0? yes	Left shift
101111	10100_	4		a+m
111001	10100_	4	a<0? yes	bit=0
111001	101000	3	a<0? yes	Lett shift
110011	01000_	3		a+m
111101	01000_	3	a<0? yes	Bit =0
111101	010000	2	a<0? yes	Shift left
111010	10000_	2		a+m
000100	10000_	2	a<0? no	Bit 1
000100	100001	1	a<0? no	Shift left
001001	00001_	1		a-m
111111	00001_	1	a<0? yes	Bit =0
111111	000010	0	stop	

			a<0? yes	a+m
001001	000010			

Remainder= A = 9 = 001001
Quotient= Q = 2 = 000010
Correct.

Division of signed numbers

Algorithm:

1. Load the divisor into the M register and the dividend into the A, Q registers. The dividend must be expressed as a 2n bit twos complement number. If MSB of Q is 1, all n bits in A are 1. Same for 0.
2. Shift A and Q one bit to the left
3. If M and A have the same signs, perform A = A-M.
4. If M and A have different signs, perform A=A+M
5. The preceding operation is successful if the sign of A is the same before and after the operation.
   A. if operation is successful and A=0, set last bit of Q=1
   B. If operation is unsuccessful and A!=0, set last bit as 0. Restore the prev value of A.
6. Repeat steps 2 thru 5 as many times as there are bit positions in Q
7. The remainder is in A. If the signs of the divisor and dividend were the same, then the quotient is in Q. Else, the correct quotient is the two's complement of whatever value is in Q.

Questions

1. 7/-3
   Divisor = -3 = M
   3 = 0011
   1100
   1
   1101 = -3 =M
   Divident=7=Q
   Q = 0111

A	Q	n	Verdict	Task
0000	0111	4		shift
0000	111_	3	Different signs	A+M
1101	111_		Unsuccessful,	Set 0, restore

			diff signs	
0000	1110			shift
0001	110_	2	Diff signs,	A+M
1110	110_		unsuccessful	Set 0 restore
0001	1100			shift
0011	100_	1	Diff signs	a+M
0000	1001		Same sign	Set 1
0000	1001			shift
0001	001_	0	Diff sign	a+m
1110	001_		Diff sign	Set 0 restore
0001	0010			stop

Remainder = 0001 = 1
Different signs of M and Q: Take 2's complement for answer
0010 ->
1101
+1
1110 -> answer = Quotient = -2

D = Q*V+R //important to show
7 = (-2*-3)+1 = 7

2. -7/3
   Divisor = 3 = M
   3 = **0011**
   Divident=-7
   0111
   1000
   +001
   1001
   Q = 1001

A	Q	n	Verdict	Task
1111	1001	4	shift	
1111	001_		Different sign	A+M

0010	001_		Diff sign	Set 0, restore
1111	0010	3	shift	
1110	010_		Diff sign	a+m
0001	010_		Diff sign	Set 0, restore
1110	0100	2	shift	
1100	100_		Diff sign	a+m
1111	100_		Same sign	Set 1
1111	1001	1	shift	
1111	001_		Diff sign	a+m
0010	001_		diff	Restore, 0
1111	0010	0		stop

Remainder = 1111 = -1
Different signs of M and Q: Take 2's complement for answer
0010 ->
1101
+1
1110 -> answer = Quotient = -2

D = Q*V+R //important to show
-7 = (-2*3)-1 = -7

3. -7/-3
   Divisor = -3 = M
   3
   0011 = -M
   1100
   +1
   1101 = M
   Divident=-7
   0111
   1000
   +001
   1001
   Q = 1001

A	Q	n	Verdict	Task

1111	1001	4	shift	
1111	001_		Same sign	a-m
0010	001_		Diff sign	Restore, 0
1111	0010	3	shift	
1110	010_		Same sign	a-m
0001	010_		Diff sign	Restore 0
1110	0100	2	shift	
1100	100_		Same sign	a-m
1111	100_		Same sign	Set 1
1111	1001	1	shift	
1111	001_		Same sign	a-m
0010	001_		Diff sign	Restore 0
1111	0010	0	stop	

Remainder = 1111 = -1
Same sign, take ans as it is
Quotient = 2

D = Q*V+R //important to show
-7 = (2*-3)-1 = -7

Booth's Recoding

Algorithm

1. Calculate table of M
2. Solve for reduced value of Q :
   10 = -1
   01 = 1
   00/11 = 0
3. Perform M*Q

Questions

1. M*Q
M = 5 = 0101

Q = 0100 = 4

iteration	A	M	Task
0	0000	0000	Set both 0
+M	0000	0101	Set 0 and M
-M	1111	1011	Take 2's complement
+2M	0000	1010	Left shift of +M
-2M	1111	0110	Left shift of -M. Unset bit always 0

Q	Q-1	Reduced val of Q	Task details
0100	0	+1,-1,0,0	Pairing ltr: 01,10,00,00
		1 -1 , 0 0	2(first)+second
		2(1)+-1 , 2(0)+0 = 1,0	
		Q = 1, 0	Perform M*Q binary multiplication 0101 *10 10100 = 20

2. 9*-6
   M = 01001
   Q = 00110's complement
   Q = 11010 = -6

iteration	A	M	Task
0	00000	00000	Set both 0
+M	00000	01001	Set 0 and M
-M	11111	10111	Take complement

+2M	00000	10010	Left shift of +M
-2M	11111	01110	Left shift of -M. Unset bit always 0

Q	Q-1	Reduced val of Q	Task details
11010	0	0,-1,1,-1,0	11,10,01,10,00: closed pairing
		0, -1 1, -1 0	pairing
		0, -1, -2	2(first)+second

//multiply with each of these RTL
M*Q - > 01001
*0,-1-2
(-2 into 01001 =01110)
(-1 into 01001 = 10111)
(0 into 01001 = 00000) //from table
Add but add extra 2 bits after every: 1111101110 + 11110111xx + 00000xxxx
= 111001010
Take 2's complement since Q and M ka sign is different
110110 = 54

3. 15*-10
M = 01111
Q = 01010's complement
Q = 10110 = -10

iteration	A	M	Task
0	00000	00000	Set both 0
+M	00000	01111	Set 0 and M
-M	11111	10001	Take complement
+2M	00000	11110	Left shift of +M
-2M	11111	00010	Left shift of -M. Unset bit always 0

Q	Q-1	Reduced val of Q	Task details
10110	0	-1,1,0,-1,0	10,01,11,10,00 closed pairing
		-1, 1 0, -1 0	pairing
		-1, 2, -2	2(first)+second

//multiply with each of these RTL
M*Q - > 01111
*-1,2,-2
-1*M = 10001
2*M = 11110
-2*M = 00010
Add but add extra 2 bits after every: 10001xxxx + 11110xx + 00010
= 1101101010
Take 2's complement since Q and M ka sign is different
10010110 = 150

Floating Point IEEE 754 Representation

Single precision: 32 bits : 1 bit for sign, 8 bits for biased exponent, and 23 bits for fraction/mantissa
Double: 64 bits: 1 sign , 11 biased exponent, 52 fraction/mantissa

Steps:

1. Convert to binary
2. Normalization: Rewrite step 1 into 1.n form
   Example: 111.011 = $1.11011 * 2^2$
   Ex: 0.00010 = $00001.0 * 2^{-4}$
3. Biasing
   Applying single precision: E-127 And double precision E-1023 on exponent step 2
4. Representation in single 32 bit and double 64 bit format.

Questions

1. **12.25**

12 = 1100 //show steps
25 = .01 //show steps

12.25 = 1100.01
<shift decimal to get 1. Something format>
$1.10001 * 2^3$

Single precision biasing:
E’ = E - 127
E’=3
E = 130
Double precision biasing
E’ = E-1023
E’=3
E=1026

Find binary of:
130 -> 1000010
1026 -> 1000000010

Single

Sign	Biased Exponent 8 bits	Mantissa 23 bits
0	10000010	1000100000000000…

Double

Sign	Biased Exponent 11 bits	Mantissa 52 bits
0	10000000010	1000100000000000000…

2. 25.44

25 = 11001 //show steps
44 = .011100001010001… //show steps

25.44 = 11001.01110
<shift decimal to get 1.smthing>
$1.1001011100001010001 * 2^4$

Single precision biasing:
E’ = E - 127
E’= 4
E = 131
Double precision biasing
E’ = E-1023
E’= 4
E=1027

Find binary of:
131 -> 1000011
1027 -> 1000000011

Single

Sign	Biased Exponent	Mantissa
0	10000011	1001011100001010001

Double

Sign	Biased Exponent	Mantissa
0	10000000011	1001011100001010001

3. **0.00635**

0 = 0
.00635 = .000000011010000

0.00635 = 0.000000011010000
<shift decimal to get 1.smthing>
$1.1010000 = 2^{-8}$

Single precision biasing:
E’ = E - 127
E’= -8
E = 119
Double precision biasing
E’ = E-1023
E’= -8
E= 1015

Find binary of:
119-> 1110111
1015 -> 1111110111

Single

Sign	Biased Exponent	Mantissa
0	1110111	1010000

Double

Sign	Biased Exponent	Mantissa
0	1111110111	1010000

4. **263.3**

263 = 100000111
.3 = 01001100110011001…

0.00635 = 100000111.01001100110011001…
<shift decimal to get 1.smthing>
$1.00000111010011001100110011001 = 2^8$

Single precision biasing:
E’ = E - 127
E’= 135

Double precision biasing
E’ = E-1023
E= 1031

Find binary of:
135 -> 10000111
1031 -> 1000000111

Single

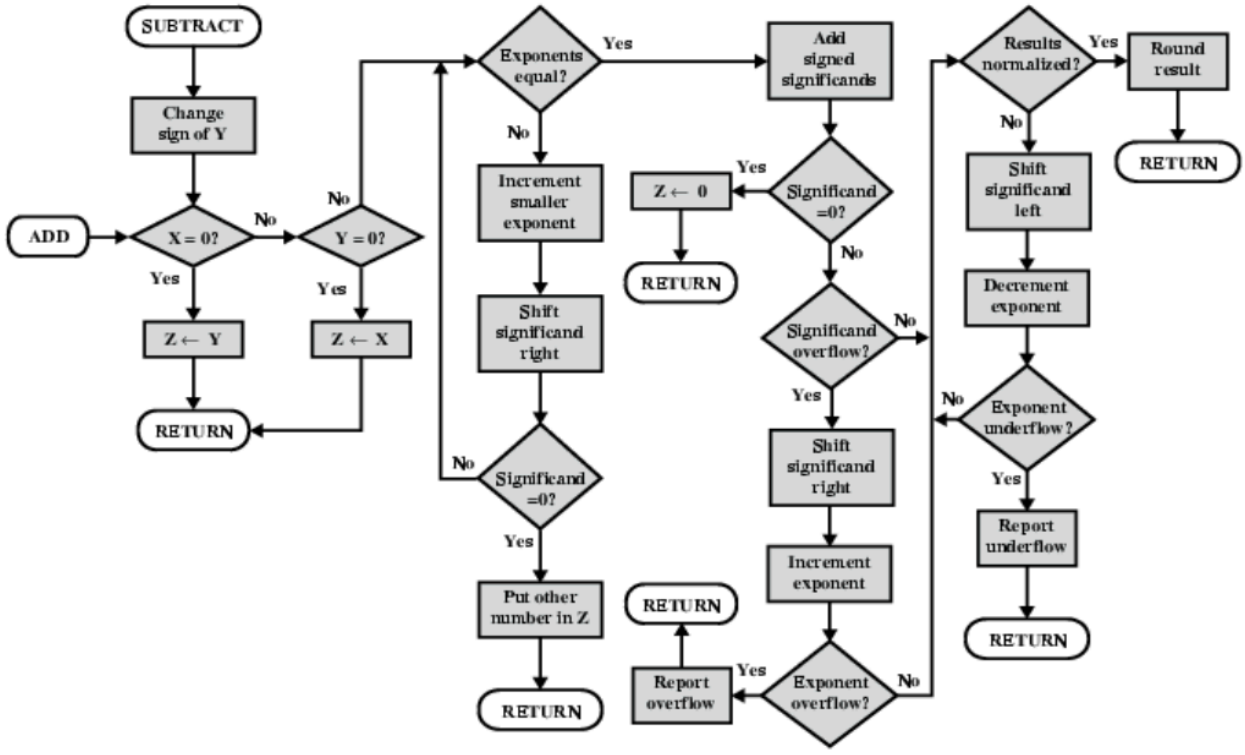
Sign	Biased Exponent	Mantissa
0	10000111	$00000111\overline{1001}$

Double

Sign	Biased Exponent	Mantissa
0	1000000111	$00000111\overline{1001}$

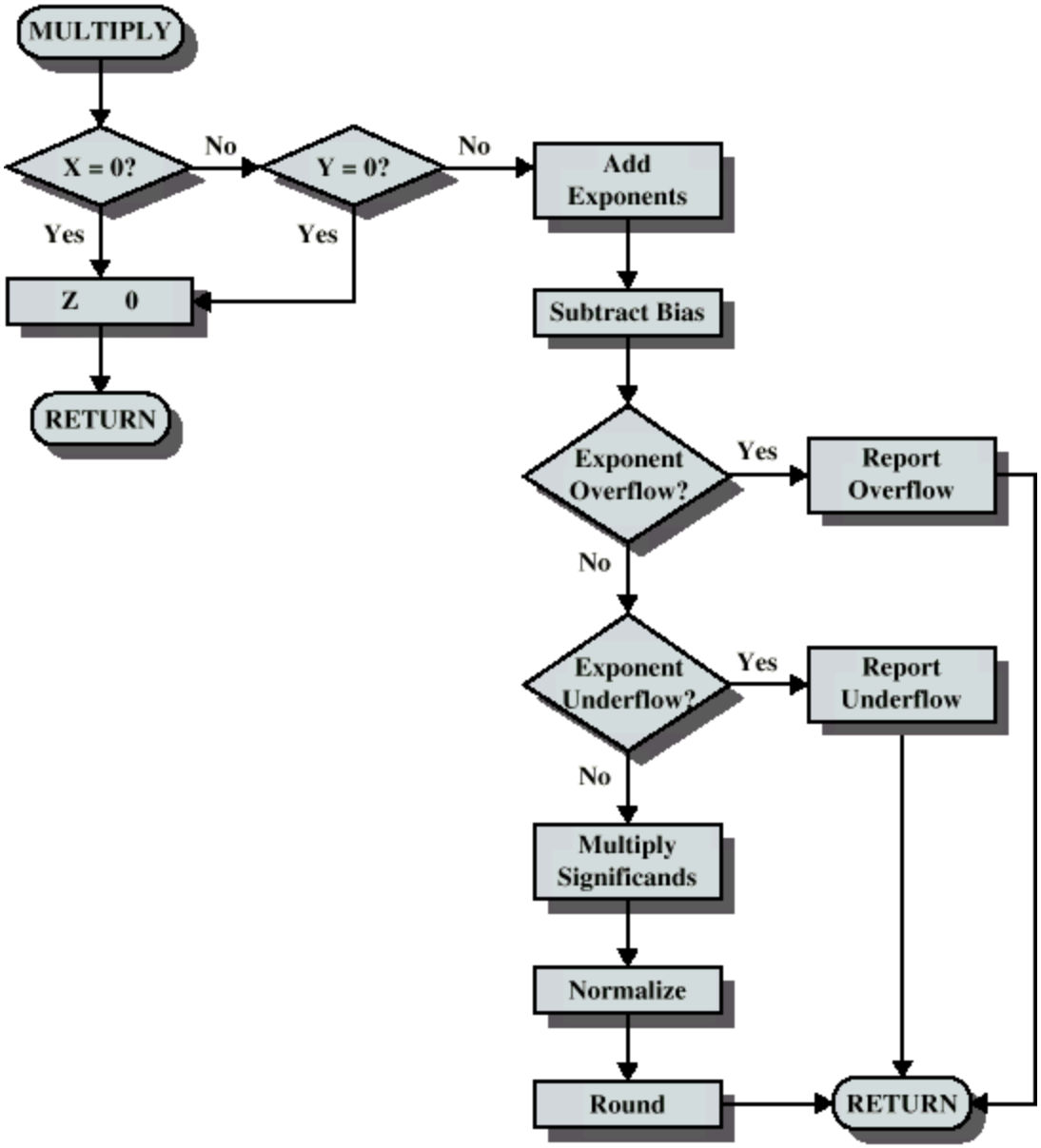
Floating point addition

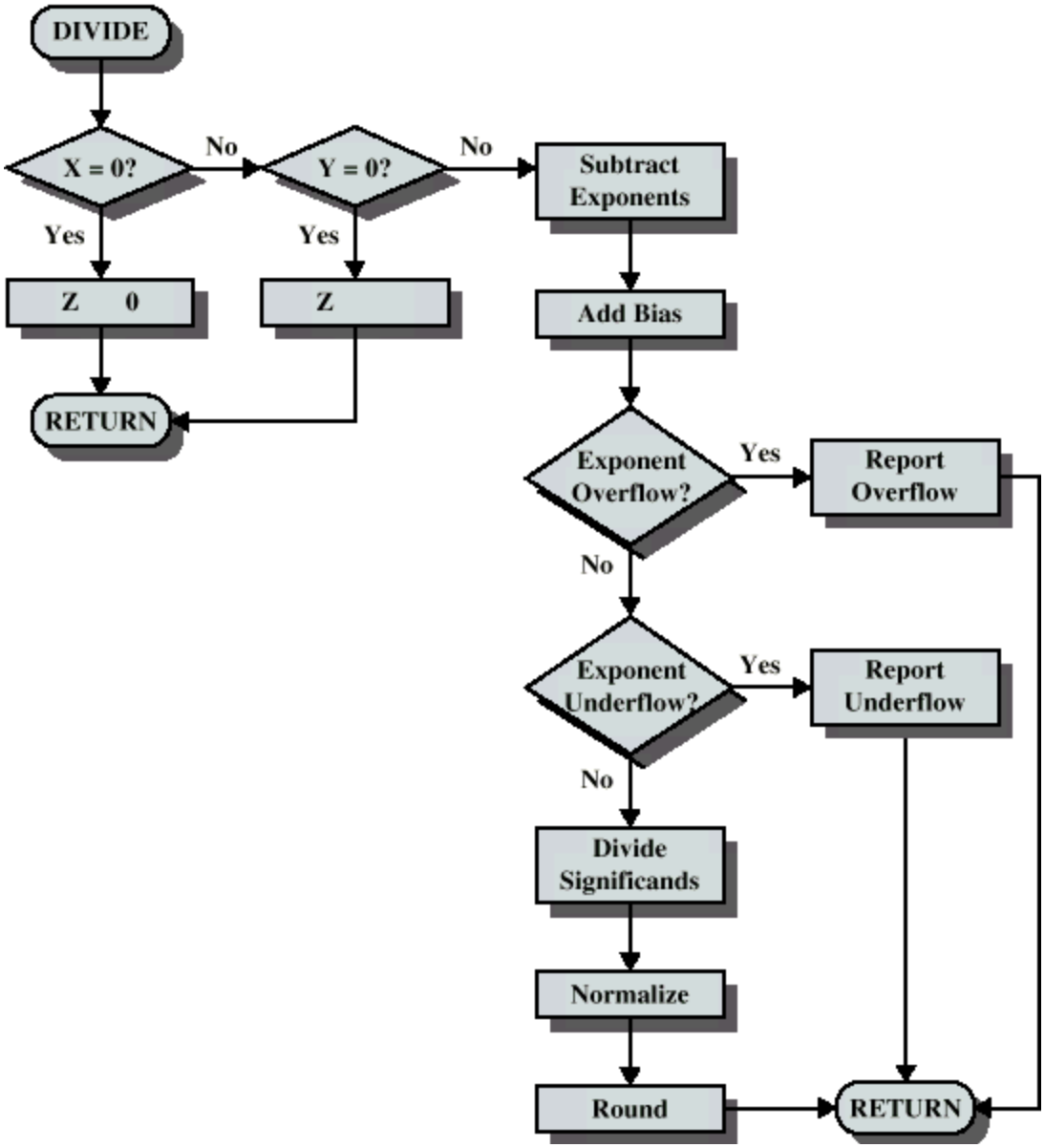
1. Rewrite the smaller number in the format of the larger number wrt exponent
2. Add the mantissas
3. Perform normalization: bring answer to 1.something form



Floating point multiplication and division

Prolly aint coming

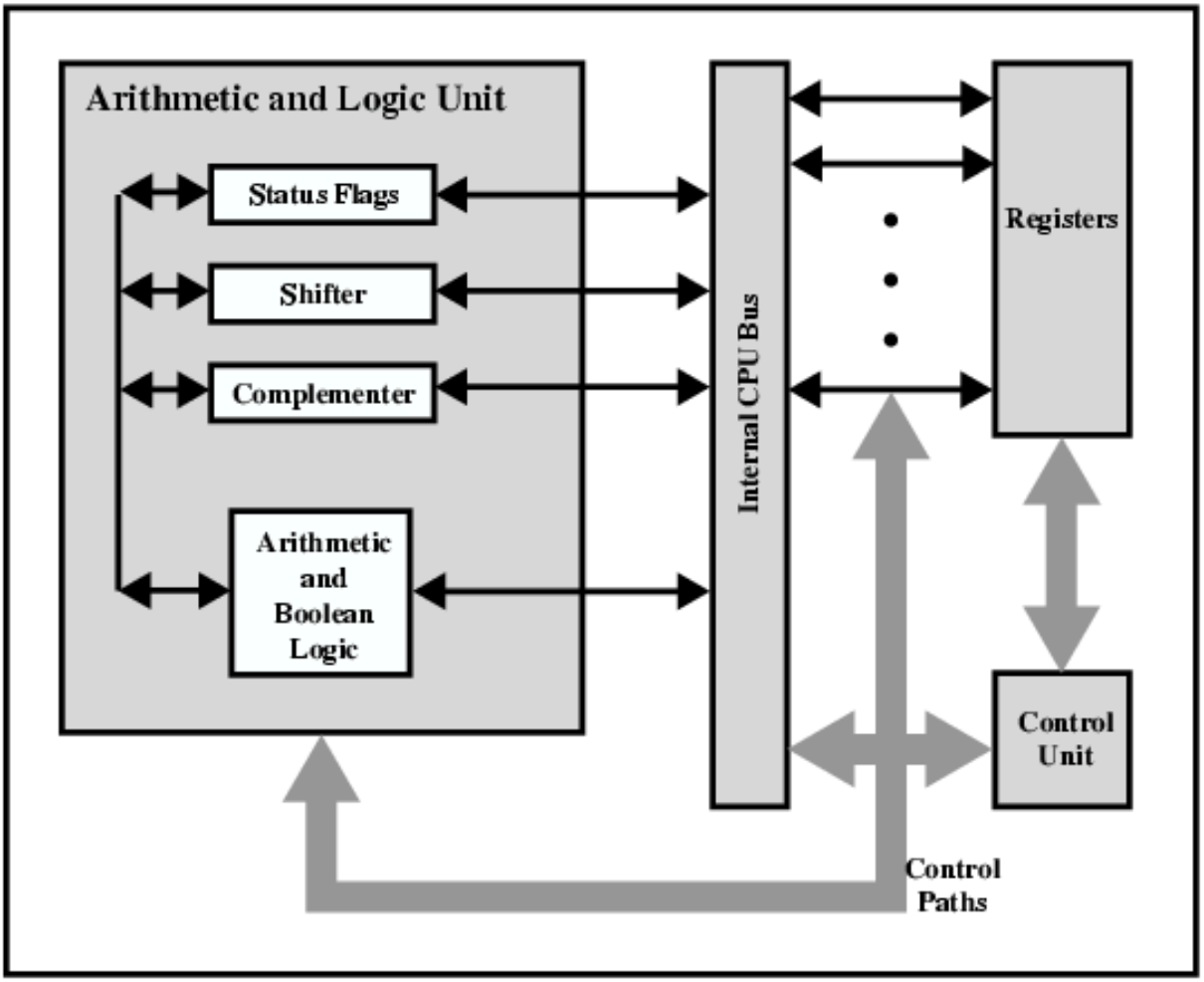

MULTIPLY
X = 0?
No
Y = 0?
No
Add Exponents
Yes
Yes
Z 0
Subtract Bias
RETURN
Exponent Overflow?
Yes
Report Overflow
No
Exponent Underflow?
Yes
Report Underflow
No
Multiply Significands
Normalize
Round
RETURN


DIVIDE
X = 0?
No
Y = 0?
No
Subtract Exponents
Yes
Yes
Z 0
Z
Add Bias
RETURN
Exponent Overflow?
Yes
Report Overflow
No
Exponent Underflow?
Yes
Report Underflow
No
Divide Significands
Normalize
Round
RETURN

Module 3: Control Unit

Control Unit

Internal structure of a CPU



Must have functions of a CPU

1. Fetch instructions
2. Interpret instructions
3. Fetch data
4. Process data (apply arithmetic/logical operations on data)
5. Write data

Registers

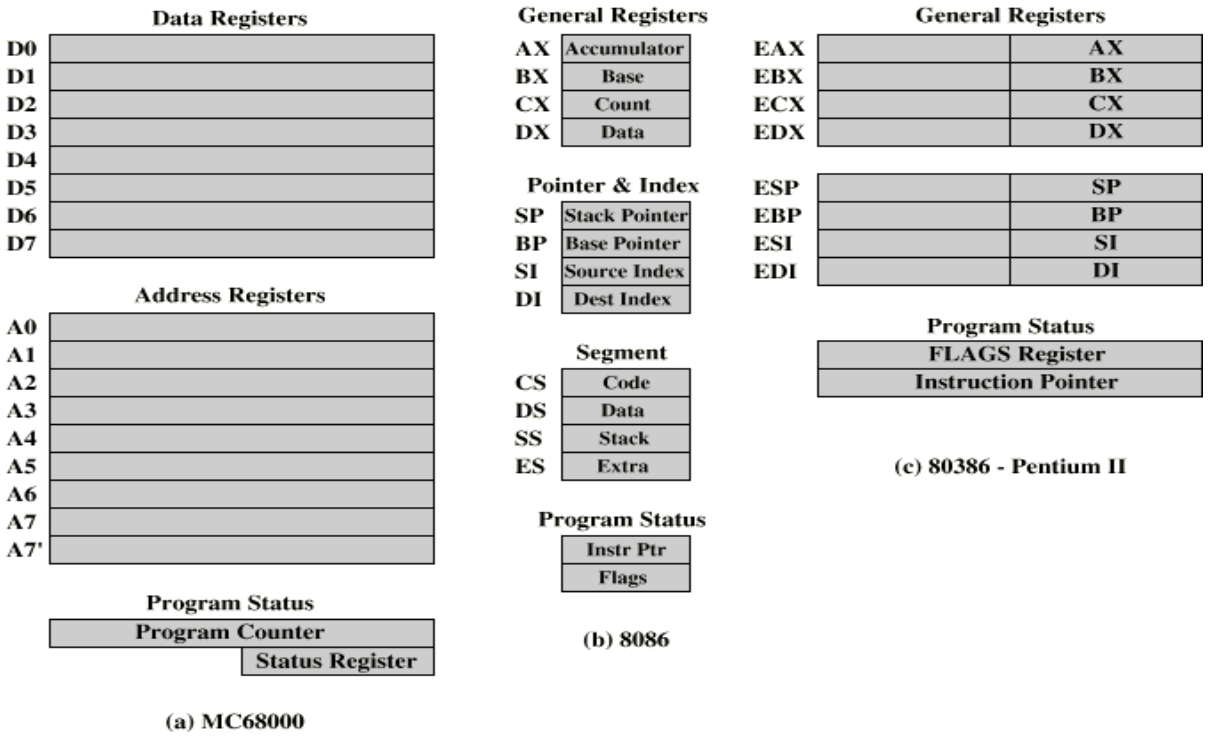
CPUs must have some temporary storage that works as their working space. This temporary storage is called registers. The number and functions of registers varies among processor designs. The type of register used is a huge design decision, and is at the top level of memory hierarchy.

User visible registers:

- Enable the machine to minimize references to the main memory by optimal use of registers.
  When registers are present, instructions don't need to access the main memory every time for input: they can just take it from the registers if they hold that data.
- Can have a variety of functions
- Data: Used only to hold data.
  A constant like #5 in **MOV R0, #5**. Here, #5 is data-only and not stored in a register for address calculation.
- Address
  - Segment pointers: hold the base address of a segment
  - Index register: stores indexed address. May increase/decrease
  - Stack pointer: Points to the top of stack
  - Control/Status registers
  - Used by the control unit to control the operation of the processor. Operating system programs control the execution of programs.

Types:

1. Data registers
2. Address registers
3. General registers



In the diagram, right hand side, E stands for extended in every abbreviation. EAX is the 32-bit extension of the accumulator register (AX), and it's a core general-purpose register in x86 architecture.

//for specific registers, only remember what component registers exist and an overview of what they do. No need to remember bits wagerah.

Mc680000: motorola 68k processor

1. **Data registers** (D0–D7): 8 × 32-bit, mainly for data operations; can be used as index registers in addressing.
2. **Address registers** (A0–A7): 8 × 32-bit, hold memory addresses; A6 = user stack pointer, A7 = supervisor/OS stack pointer. //just remember ek is for stack and ek is for os stack. Kaunsa hai doesn't matter.
3. **Program counter (PC):** 32-bit, holds the next instruction's address.
4. **Status register (SR):** 16-bit, stores flags and processor state.

Intel 8086

1. **Data registers** (AX, BX, CX, DX): 16-bit, general-purpose.
2. **Pointer/index registers** (SP, BP, SI, DI): 16-bit, used for stack, memory, and indexing.
3. **Segment registers** (CS, DS, SS, ES): 16-bit, hold segment base addresses.

4. **Instruction pointer** (IP): 16-bit, points to next instruction
5. **Flags register**: 1-bit status and control flags.

80386 Pentium

An extension of the 8086.

1. **General-purpose registers:** 8 × 32-bit (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP).
2. Retains all registers of Intel 8086.

General Purpose Registers

1. May be general purpose or restricted.
2. Can be used for data or addressing.
3. Types include:
    a. **Data registers(D0 to DN)**: holds value for data manipulation
    b. **Accumulator(AX)**: specially optimized for arithmetic/io
    c. **Addressing registers(A0 to AN)**: holds addresses
    d. **Segment registers(CS,DS,SS,ES)**: hold base address of memory segments
4. Why make them general purpose?
    a. Increases flexibility
    b. Increases instruction size (one register can do the work jo earlier many registers used to do) but also increases complexity.
5. Why make them specialized?
    a. Smaller (and faster) instructions.
    b. Less flexibility.
6. AX is the primary accumulator. The CPU prefers AX for operations like ADD, SUB, MUL, DIV. Many instructions automatically use AX without you specifying it. It is the default register for I/O and arithmetic.
7. BX is the base register. Used in indexed addressing. What is indexed addressing? If u know the base address to be M and u know the index of your element is 5, the address of your element will be M+5. This is called indexed addressing.
8. CX is called count register. It stores the loop count in iterative operations.
9. DX is called data register and is used in input output operations. Used with AX in multiplication and division ops.

- Registers are large enough to hold a full address. Registers are large enough to hold a full word. Often possible to combine two data registers (e.g., in C programming, long int a.)

- **Status registers:** holds the sets of individual bits that tell if an operation was successful.
    - Cannot be read implicitly by programs
    - Cannot be set directly by programs
- Some applications: sign bit when representing signed numbers, flag conditions in certain algorithms, condition codes, if this do that etc, equal flag, and so on

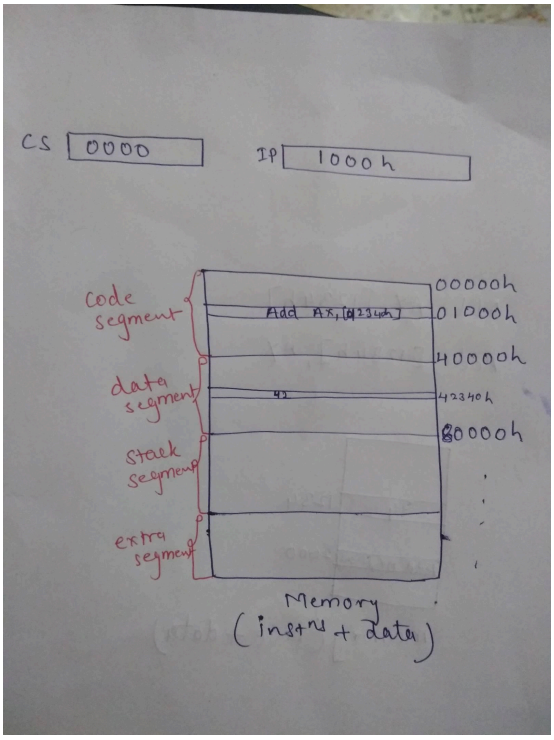
Important Control Unit Registers:

1. **Program Counter (PC):** Holds the address of the next instruction to be executed.
2. **Instruction Register (IR):** Holds the current instruction being executed.
3. **Memory Address Register (MAR):** Holds the memory address to be accessed.
4. **Memory Buffer Register (MBR)**: Holds the data being transferred to or from memory.

Pointer Registers

IP (Instruction pointer)

1. 16 bit
2. Stores offset address of next instruction
3. IP and CS ka association (CS:IP) gives complete address of current instruction



Stack pointer

1. 16 bit
2. Provides offset value within the program stack
3. (Stack segment) SS:SP refers to the current position of data/address within the stack

Base pointer

1. 16 bit
2. Helps in referencing the parameter variables passed to subinstruction.
3. Address in SS(stack) is combined with offset in BP to locate the parameter.

4. Can also be paired with DI and SI for special addressing.

Index Registers: SI and DI

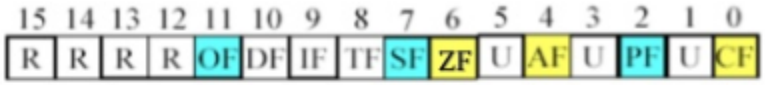
1. Source index: used as source for string ops
2. Destination index: Used as destination index for string ops

Control registers

1. 32 bit instruction pointer register + 32 bit flag register = control register
2. Instructions involve comparisons and calculations
3. They also change the status of flags

Flag register

1. Also called status register
2. 16 bit
3. Conditional flags include CF, PF, AF, ZF, SF, OF
4. Control flags include TF,IF,DF



R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag

SF = sign flag
ZF = zero flag
AF = auxiliary carry flag
PF = parity flag
CF = carry flag

- **Overflow Flag:** indicates the overflow of a high order bit after a signed operation.
  OF=1, signed result is incorrect
- **Direction Flag:** determines the left or right direction for moving and comparing string data. df=: operation takes left to right direction. When value=1, operation takes rtl
- **Sign flag:** represents if a number is positive or negative by altering the MSB. SF=0, number is positive. SF=1, negative.
- **Trap flag:** allows setting the operation of processor in single step mode. Used for debugging. By setting the TF, a debugger can pause the program after every single instruction. TF=1: debugger can pause.

- **Zero flag:** Indicates the result of an arithmetic operation or comparison operation. Nonzero result, flag 0. Zero result, flag is 1.
- **Interrupt flag:** determines whether external interrupts like keyboard mouse etc are to be considered or ignored. Disables interruption when flag is 0, allows at 1
- **Auxiliary carry flag:** Contains carry from bit 3 to bit 4 in arithmetic operation.
- **Parity flag:** after an arithmetic operation, if the number of set bits is even, PF=1 and if odd, PF=0.
- **Carry flag:** contains the carry 0 or 1 from an arithmetic operation. Also stores content of last bit of a shift operation. >>, <<.

Difference between ACF and CF: AC = nibble carry (bit3→bit4) and CF = full carry (MSB).
Mnemonic: Oily dishes in Telangana stay zeroed under arms of penguins and cats.

Segment Registers

Segments are specific areas for containing data, code, and stack.

1. **Code segment:** contains all instructions/code. 16 bit ka code segment register stores the starting address of the code segment
2. **Data segment:** Contains data, constants, and work areas. Stores the starting address of data segment
3. **Stack segment**: Contains the data and return addresses of procedures or subroutines. Implemented as a stack and stores the starting address of the stack.
4. **Extra Segment**

Addressing Modes

- Addressing mode tells how to find the operand.
- Operand can be in the accumulator, a general purpose register, or a memory location.
- Opcodes may use different addressing modes. In the instruction format, a few bits indicate the mode. The value of these bits tell which addressing mode to use.
- Without virtual memory = effective address is a register or a main memory location
- With virtual memory= effective address is a register or a virtual address.

Instruction Addressing Terms

- A (Address field value): The number written in the instruction.
  Example: LOAD [500] → A = 500
  Load the value at memory address 500 into a register.
  LOAD R1,[500]: LOAD memory 500 into register R1.
  No register mentioned = default register (AC)
  Applies to all^^

- R (Register field value): The number pointing to a register.
  Example: LOAD R1 → R = 1 → register R1
  Load the value from register R1 into another register.

- EA (Effective Address): The actual location of the operand.
  CPU uses it to fetch or store data.
  Example: LOAD R1, 100(R2)
  EA = R2 + 100
  LOAD 100(R2)

- (X) (Contents at X): The value stored at memory/register X.
  Example: If memory[500] = 25, it implies that (500) = 25
  The contents of memory location 500 is 25; (500) represents the value stored there.

- **STORE 500** → Store the value from a register into memory address 500.
  STORE R1, 500

- **ADD 500** → Add the value at memory address 500 to a register.

- **ADD R1** → Add the value in register R1 to a register.

- **MOV R1, R2** → Copy the value from register R2 to register R1.

- **MOV 500, R1** → Copy the value from register R1 into memory address 500.

- **SUB 500** → Subtract the value at memory address 500 from a register.

- **SUB R1** → Subtract the value in register R1 from a register.

8086 Real Mode Addressing Modes:

Note: EA = effective register

1. **Immediate Addressing:** Operand is in the instruction itself.
    - Example: ADD 5 → adds 5 directly.
    - Fast, no memory fetch, but operand size limited.
2. **Register Addressing:** Operands in CPU register.
    - EA = register itself.
    - Example: MOV A, B → move B to A.
    - Fast, short instructions.
3. **Memory Addressing:** Operand in memory.
    - **Direct Memory:** Instruction gives memory address.
        - Example: ADD AX, [123h].
        - Single memory reference, limited address space.
    - **Register Indirect:** Register holds operand address.
        - Example: MOV [SI], AL.
          SI is the register (source index)
        - Saves instruction space, flexible for arrays/pointers.
    - **Based:** Base register + displacement.
        - Example: MOV AL, [BX+5].
        - Useful for structs/stack frames.
    - **Indexed:** Index register + displacement.
        - Example: MOV AL, [SI+5].
        - Signed displacement allowed.
    - **Based + Indexed:** Base + Index (optional displacement).
        - Example: ADD AX, [BX+SI] or [BX+SI+8].

    - **String Addressing:** Uses SI (source) & DI (dest).
        - Example: MOVSB.
        - Stands for move string byte

4. **Stack Addressing:** Operand at top of stack (PUSH/POP).

5. **Auto-increment/Decrement:** Register provides address, then updates automatically.

Protected Mode (80386+):

- **EA (Effective Address):** Offset inside a segment
- **Linear Address:** Segment base + EA.
- Paging is the process that converts **linear → physical address**.
- Supports **12 addressing modes** (Immediate, Register, Displacement, Base, Base+Displacement, Scaled Index+Displacement, Base+Index, Base+Index+Displacement, etc.).

Quick skim: Addressing modes

Addressing Mode	EA Formula / Explanation	Example Instruction	Usage / Notes
Immediate	Operand is in instruction itself	`MOV AX, 2000`	Fast; no memory access; operand size limited.
Direct / Absolute	EA = Address field in instruction	`MOV AX, [1234H]`	Instruction gives exact memory location; single memory access; simple.
Register	EA = Register itself	`MOV AX, BX`	Operand is in CPU register; very fast; short instruction.
Register Indirect	EA = contents of register	`MOV AX, [BX]`	Register holds memory address; flexible for arrays/pointers; 1 reg + 1 mem access.
Memory Indirect	EA = memory[Address field]	`MOV AX, [[5000H]]`	Instruction points to memory location holding EA; requires 2 memory fetches.
Base Register / Based	EA = Base register + optional displacement	`MOV AL, [BX+5]`	Useful for structures, stack frames; flexible; still 1 memory reference.
Indexed	EA = Index register + displacement	`MOV AL, [SI+5]`	For arrays/tables; signed displacement allowed.
Base + Index	EA = Base register + Index register	`ADD AX, [BX+SI]`	Combines base and index; good for structured data access.
Base + Index + Displacement	EA = Base + Index + displacement	`MOV AX, [BX+SI+8]`	Most flexible; used for complex arrays, structs, stack frames.
Relative / PC-relative	EA = PC + displacement	`JMP 05H`	Mainly for jumps/branches; displacement relative to current instruction.
String	EA = SI source, DI destination	`MOVSB`	Automatically handles source/dest pointers; useful for string operations.
Stack / Auto Increment / Decrement	EA = SP (top of stack), auto updates	`PUSH AX` / `POP AX`	Stack operations; SP updates automatically.

Instruction cycle

What happens?

- Program counter (PC) holds the address of the next instruction to fetch.
- Processor fetches the instruction from the memory location as pointed by PC
- Increments PC (unless otherwise specified)
- Instruction is loaded into the Instruction Register IR
- Processor interprets the instruction and performs the necessary actions.

Step-wise cycle of how instruction cycle runs.
T1: MAR <- PC
Copy the program counter into the memory address register.

T2: MBR(memory buffer register) <- MAR
Memory is read at the address stored in MAR
Fetched word or info is placed into MBR, which temporarily stores instructions.

T3: PC <- Pc + I and IR <- MBR
Pc increments by instruction length I
Next instruction address is ready.
Instruction is moved from MBR to IR
CPU decodes instruction from IR
These micro operations like transfer (MAR <- PC), arithmetic (PC <- PC+1), memory, register load etc. depend on CPU design but largely:

1. Fetch
   PC -> MAR → Memory → MBR → IR → PC+1
   a. Program counter contains address of next instruction
   b. This address is moved to Memory address register
   c. This address is placed on the address bus
   d. Control Unit requests memory to be read
   e. Result is placed on data bus
   f. From data bus, data is copied to memory buffer register
   g. Then data goes to Instruction Register
   h. Program counter increases by one instruction length.
2. Processor memory
   a. Data transfer between CPU and main memory
3. Processor IO
   a. Data transfer between CPU and io module
4. Data processing

    a. Some arithmetic operations or logical operations on the data
5. Control
    a. Alteration of sequence of operations, like using break etc
6. Combination of all

Working of an Add Instruction

Ex: ADD R1, 500

- T1: MAR ← IR(address)
    - Instruction Register (IR) holds the instruction.
    - The address part of instruction is copied to Memory Address Register (MAR).
    - Tells memory where operand is.
- T2: MBR ← Memory[MAR]
    - CPU sends MAR's address to main memory.
    - Memory fetches the operand.
    - Operand is placed into Memory Buffer Register (MBR).
- T3: R1 ← (R1) + (MBR)
    - Content of register R1 added to operand in MBR.
    - Result stored back into R1.
      <assuming R1 was the destination register provided in ADD command>

Execution Cycle

After fetching and decoding, the execute phase runs.

1. Decode instruction
   CU interprets the operation and operands.

2. Fetch operands
   CU gets required data from registers or memory.

3. Execute
   ALU performs the operation (add, move, compare, etc.).

4. Store result
   Result is written back to a register or memory.

Indirect Cycle

May require memory to fetch operands. Requires more memory access. Is like an additional instruction subcycle.

- T1: MAR ← IR(address)

- Load the address part of instruction into MAR.

- T2: MBR ← Memory[MAR]
  - Read memory at that address, put result in MBR.

- T3: IR(address) ← MBR(address)
  - Update IR's address field with the operand's actual address.

Interrupts

Types of interrupts:

1. Program: errors like divide by zero
2. Timer: from internal clock for multitasking
3. i/o: signals from input output controllers
4. Hardware failure: memory error for example

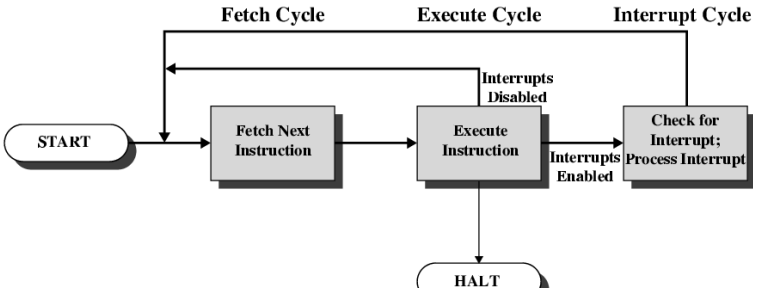
Interrupt Handling Cycle

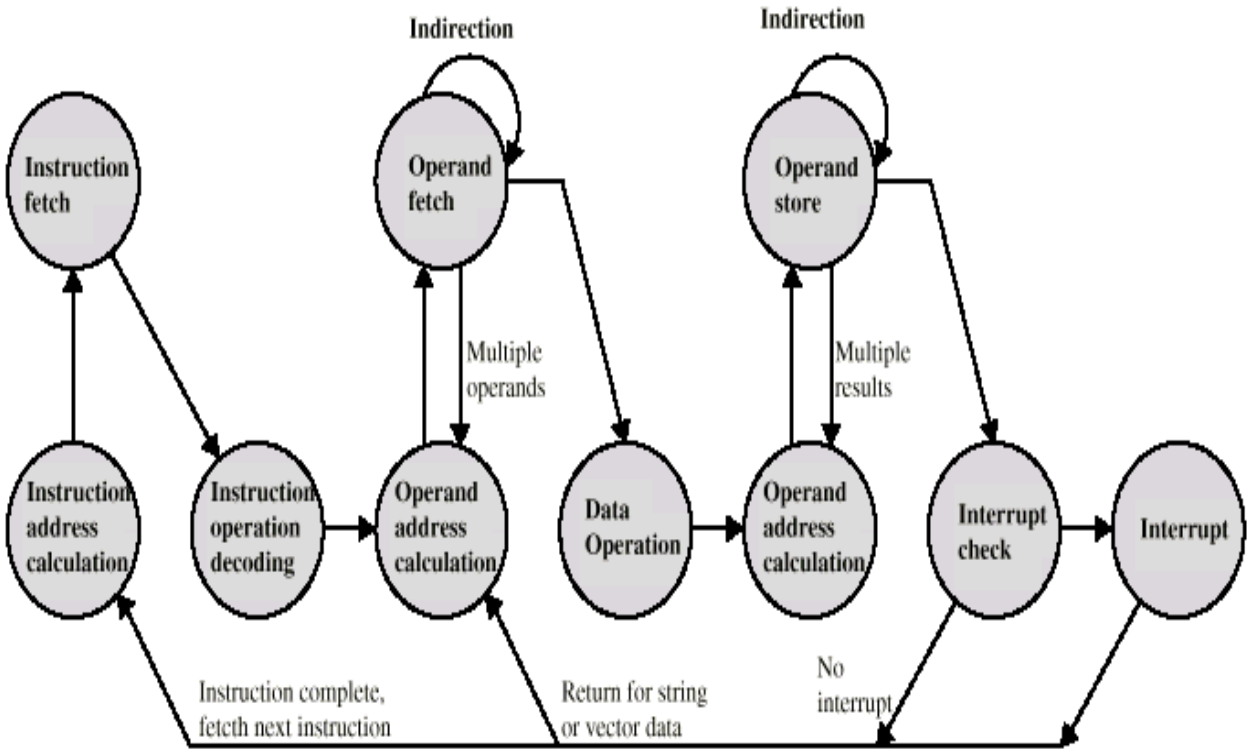
1. Check for interrupt, indicated by interrupt signal
2. If no interrupt, fetch the next instruction
3. If interrupt exists:
   a. Suspend the execution of current prg
   b. Save current program counter at a save address
   c. Set Program Counter to a routine address
   d. Process the interrupt
   e. Restore context, continue the program

T1: MBR□(PC)
T2: MAR□Save_Address
PC □Routine_Address
T3: Memory□(MBR)

Hardware Interrupt Handling

1. PC → MBR (Put the PC value into MBR)
2. SP → MAR (Stack Pointer gives the memory address where PC must be saved)
3. Memory[MAR] ← MBR (Write MBR's value (PC) into memory at MAR)
4. ISR address -> PC: Load the starting address of the Interrupt Service Routine.
5. Fetch ISR instruction
6. Begin normal fetch–execute cycle for the ISR.

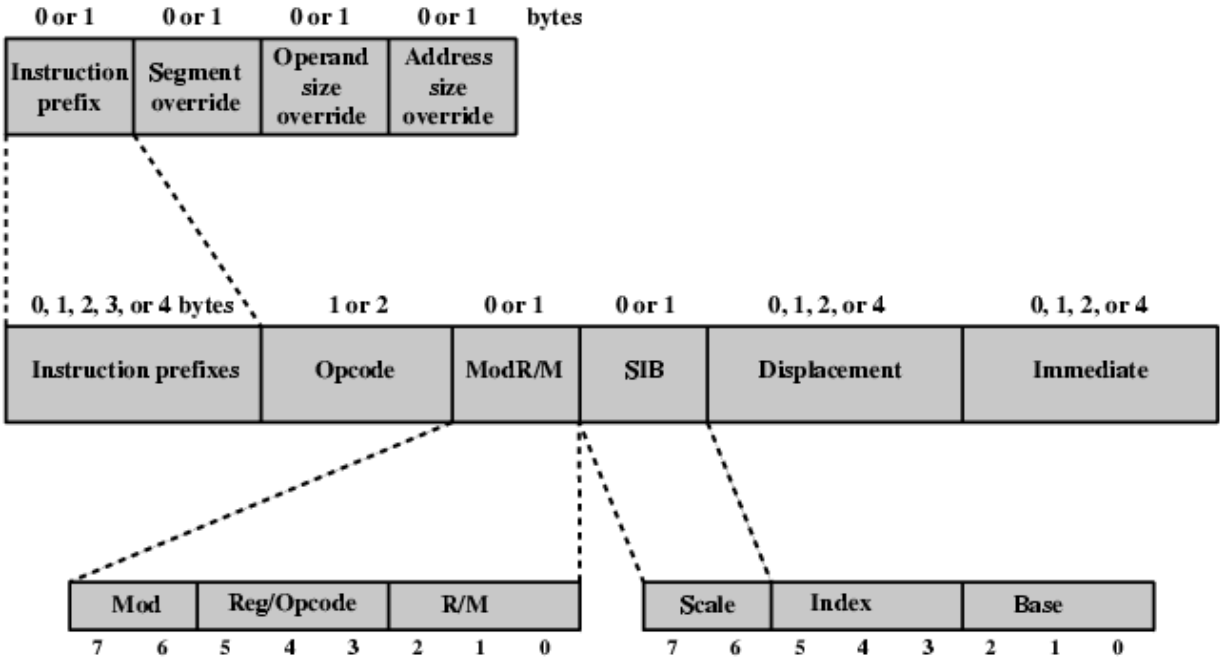




Instruction Format

- An instruction consists of an op-code(operation code like MOV, ADD, etc) and operand/s
- These could be explicitly specified in an instruction / be assumed (implicit)
- Instruction format defines bit layout
- It shows addressing mode for each operand

Depends on:
1. Memory size: larger memory size= more bits needed
2. Memory organization: virtual memory = bigger address range.
3. Bus structure: instruction length = bus width match
4. CPU speed: memory data transfer rate = processor speed
5. Number of addressing modes, operands, register sets
6. Register vs memory
7. Address range and granularity
8. Layout of bits in an instruction
9. Most instruction sets have multiple instruction formats.



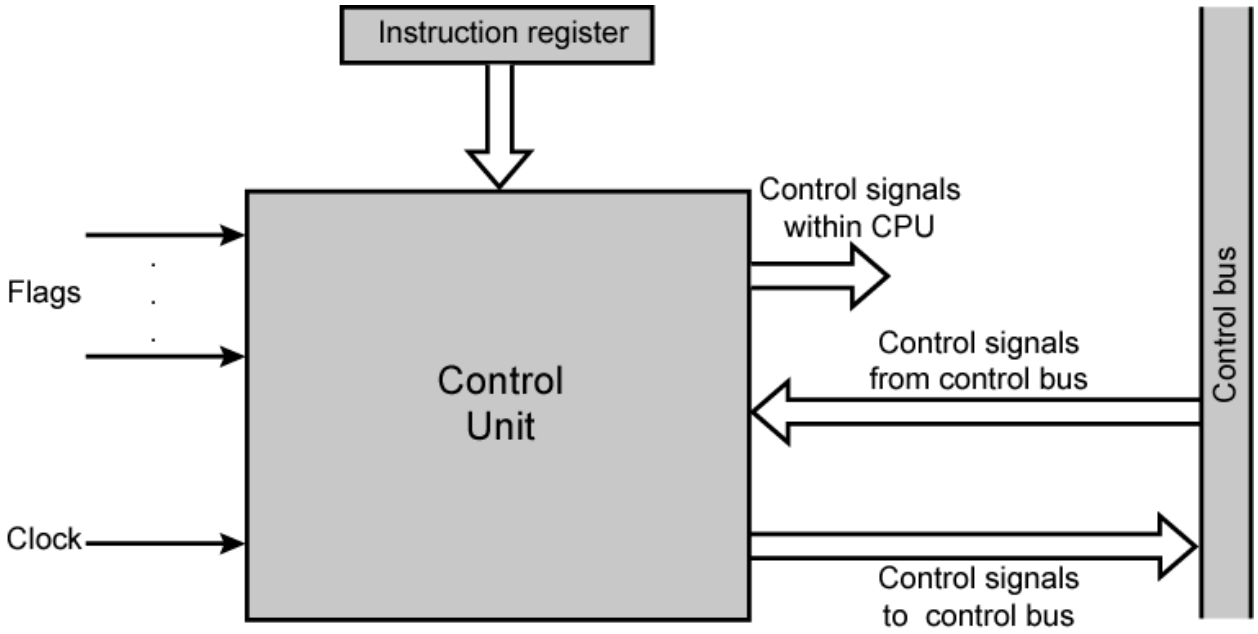
- **Instruction prefix:** LOCK (proceed without interruption) or REP (repeat some number of times) prefix.
- **Segment Override:** Explicitly specifies which segment register an instruction should use. Forces use of a specific segment (CS/DS/ES/SS).
- **Operand size:** default operand size is 16 or 32 bit. Switches between 16 or 32 bit.
- **Address size:** determines the displacement size in instructions and the size of address offsets generated during EA calculation.
- **Opcode:** Specifies if [1] data are byte or full size. [2]direction of data operation [3]whether the immediate data field should be sign extended
- **Mod R/M:** This byte tells CPU where the operand is. Register or memory mode.
- **SIB Byte: S**cale, Index, Base for complex addressing.
    a. Scale: tells CPU how much to multiply the index register by.
    b. Index: selects which index register to use.
    c. Base: selects which base register to use.
    d. Displacement: a number added to the address to reach the actual memory location.
    e. Immediate: A constant value used directly in that instruction.

CPU Execution Concepts

- **Fetch/Execute Cycle:** CPU continuously repeat fetch and execute steps to run a program.

- **Micro-operations:** Tiny atomic actions done by the CPU in a single clock cycle. The smallest building blocks of instruction execution.
- **Microinstruction:** Each microoperation is controlled by signals. A microinstruction is a set of control signals that makes a microoperation happen.
- **Microprogram:** A sequence of microinstructions that implement a single machine instruction together.
- **Microcode:** A coherent part of a microprogram.

Think of a machine instruction as a recipe.
Microprogram = list of small steps in the recipe.
Microinstruction = one small step.
Micro-operation = action done in that step (like “add 1 cup of flour”).



Functions of Control Unit

1. **Sequencing**: Causes CPU to step through a series of micro operations.
   Sequencing means the control unit decides which micro-operation should happen first, next, and last.
   It ensures every instruction is executed in the correct step-by-step order.
   Without sequencing, the CPU would perform operations randomly and programs would not run correctly.

2. **Execution**: Causes the performance of each micro operation.
   Execution means the control unit activates the exact control signals needed for each micro-operation.

It tells the ALU when to add, registers when to load, and memory when to read/write.
This ensures the micro-operations actually take place and produce results.

Control signals input

1. Clock: One micro instruction per clock cycle
2. Instruction Register: Op-code for current instruction. Determines which microinstructions are performed.
3. Flag: Stores status of the CPU and result of previous operation

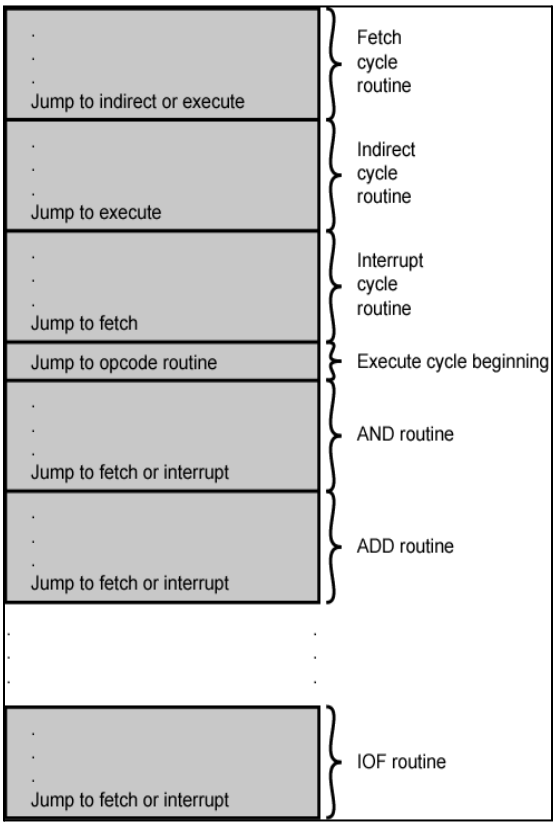
From control bus: Interruption or acknowledgement
Within cpu: causes data movement and activates functions
Via control bus: to memory or input output modules

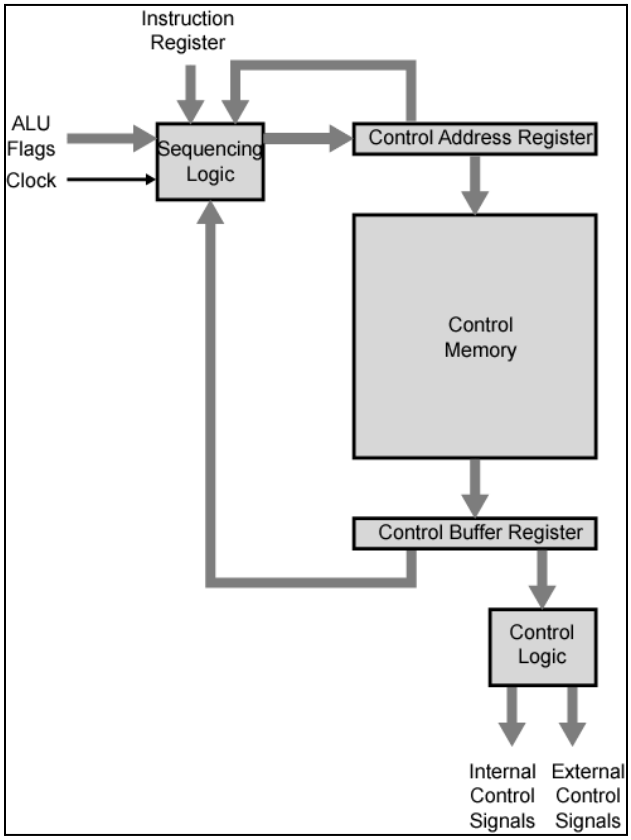
Types of control unit- 8 to 10 pnts

Hardwired CU	**Microprogrammed CU**
Speed is fast	Slow speed
Higher cost Totally hardware based	Lower cost As less hardware etc is required
Not flexible Cannot add more control signals, entire thing has to be redesigned	Flexible Easily add more control signals by adding more micro operations
Complicated design process	Systematic design process
Decoding and sequential logic is complex	Decoding and sequential logic is simple
Used in RISC	Used in CISC
Small instruction set	Large instruction size Has control field and address field, that need more ins size
Control memory is absent	Control memory present
Small chip area required	More chip area is needed: the memory needs more chip area

Implementation of Microprogrammed CU

1. The microprogrammed control unit generates a list of control signals where each signal can be on or off.
2. Every **microoperation** has an associated control word that specifies which control signals (**microsignals**) must be activated.
3. A sequence of these control words forms a **microprogram** that implements a machine code instruction.
4. The control unit uses a control address register to hold the address of the next microinstruction.
5. The control memory stores all microinstructions, and each fetched microinstruction is placed into the control buffer register.
6. The control logic then decodes this microinstruction and produces the required internal and external control signals.





Microinstruction Sequencing

Depends on ALU flags and control buffer register.
Steps for next microinstruction:

- Get next instruction.

- Increment control address register.
- Jump based on jump microinstruction.(A jump microinstruction tells the control unit to change the next microinstruction address instead of going to the next one in sequence.)
- Load address field of control buffer register into control address register.
- Jump to machine instruction routine.(after decoding the opcode, the CU jumps to the starting address of that instruction's microroutine)
- Load control address register based on opcode in IR.

Control Memory in Modern Microprocessors-skip

- Large processors → many instructions and register-level hardware.
- Many control points to manipulate → **control memory** required.
- Control memory characteristics:
    1. Contains **large number of words** (one per instruction/microinstruction).
    2. **Wide word width** due to many control points.
- Factors affecting design:
    1. Maximum number of **simultaneous micro-operations** supported.
    2. Method of **control information representation/encoding.**
    3. Method to specify **next micro-instruction address.**

Microprogramming Types-skip

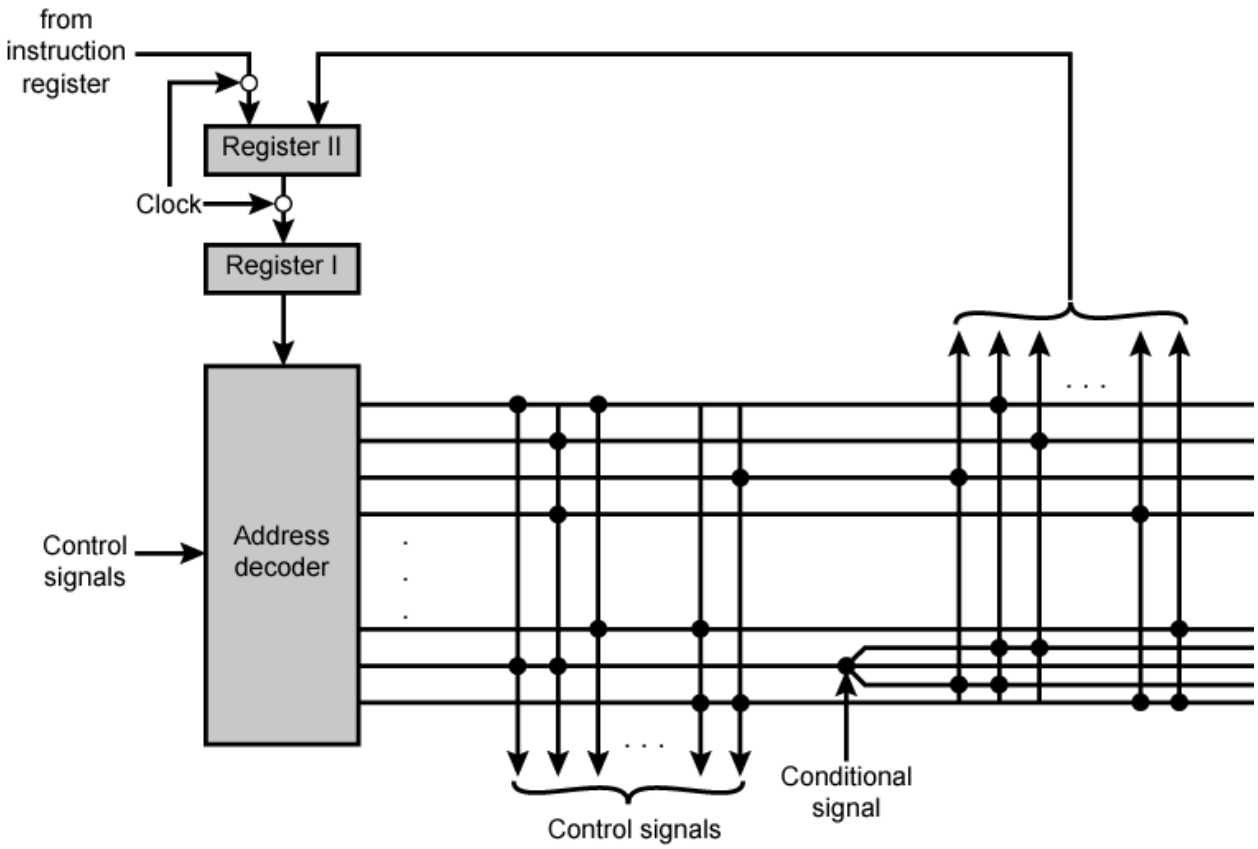
- **Vertical Microprogramming**:
    - Each microinstruction specifies a single **or few micro-operations.**
    - Narrow width.
    - Limited parallelism.
    - Requires **decoding** to generate control signals.
- **Horizontal Microprogramming**:
    - Each microinstruction specifies **many micro-operations in parallel.**
    - Wide memory word.
    - High parallelism.
    - Little encoding needed.
- **Optimized Approach**:
    - Divide control signals into **disjoint groups.**
    - Implement each group as a **separate field in memory word.**
    - Supports **reasonable parallelism** without too much complexity.

Applications of Micro Programming

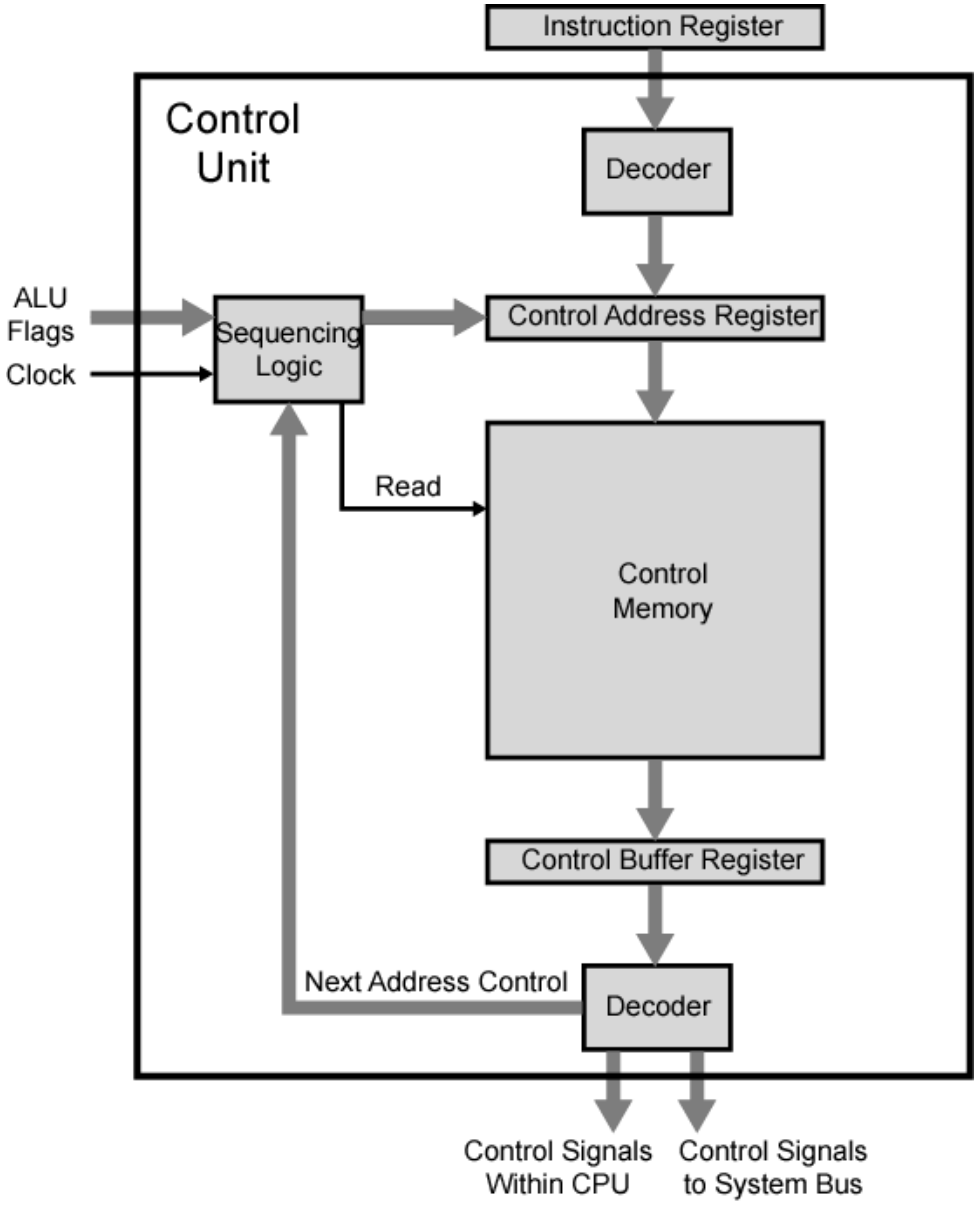
1. Implementation of control unit of computers as it is easy to implement, resize, and reconfigure
2. In emulation: Use a microprogram on a machine originally written for another system. Emulation means running a microprogram that makes your machine behave like a different machine.
3. In OS: implement some primitives of the operating system.
4. In higher level language support.
5. In microdiagnostics: To enhance system maintenance by detecting, isolating, and monitoring the repair of system errors.
6. User tailoring: By using RAM for implementation control memory, it makes it possible to customize machines to different applications.

Wilkes control overview-skip

- First microprogrammed control concept.
- Designed to simplify CPU control by replacing complex combinational logic with a microinstruction memory.
- Two main parts:
  Control Field → generates control signals for ALU, registers, memory, buses.
  Address Field → specifies address of the next microinstruction.
- Features:
  Simple and predictable.
  Hardware is static (diode matrix).
  No need for complex logic circuits to generate control signals.
  Supports sequence control via address field.
  Microinstructions executed one at a time, can build any machine instruction



It simplifies design of CU, is cheaper, and less error prone. However, it is also slower.



RISC and CISC

Intel’s hardware oriented approach is CISC and Apple’s is RISC
CISC stands for Complex Instruction Set Computer
RISC stands for Reduced Instruction Set Computer
Instruction Set Architecture(ISA): Interface to allow easy communication between the user and the hardware.
ISA means the execution, copying , editing, and deleting of data.

The primary goal of CISC is to complete the task in as few lines of assembly language as possible.

CISC features

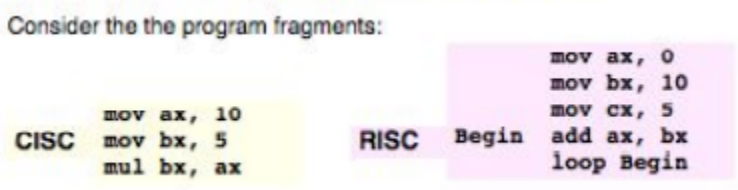
CISC = big instructions, more hardware work.

1. 120-350 instructions
2. Variable instruction and data formats
3. **Small set of general purpose registers** (8-24). These registers are used for very generic purposes like multiplications, division, etc.
4. A **large number of addressing modes**
5. **High dependency on micro program**
6. **Complex instructions can be handled**
7. **Complex pipelining**
8. Many functional chips are needed to design using CISC
9. **Difficult to design a superscalar processor**
   Superscalar architecture: Processor with higher speed. At a time, two instructions can be performed simultaneously.
10. **Rarely supports on chip cache memory**
11. Difficulty in handling data dependency, resource conflict, etc

RISC features

RISC = small instructions, more speed.

1. Instruction set with limited number of instructions
2. Simple instruction format
3. **Large set of CPU registers**
4. **Very few addressing modes**
5. **Easy to construct a superscalar processor**
6. **Complex instructions cannot be handled**
7. Hardwired CU for sequencing microinstructions
8. **Simple pipelining**
9. **Supports on chip cache memory**
10. All functional units are on a single chip. This makes the CPU faster



RISC pipelining

- Most instructions are register to register
- Two phases of execution: I and E, Instruction fetch and execute
- For load and store, IED: Instruction fetch, execute, load

Sequential Execution

One cycle at a time takes more cycles: 13 cycles here. Each column represents a cycle

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

If we optimize and perform 1’st ka execution (E) along with inputting(I) 2’nd, we reduce the number of cycles from 13 to 10. <check below table>

Two-stage pipelined timing

Note that D and E (load and execute) or D and I (input and load) cannot happen simultaneously as data cannot be handled simultaneously when there is NO CACHE MEMORY ADDED.

Load rA<- M	I	E	D							
Load rB<- M		I		E	D					
Add rC<- rA+rB				I		E–				
Store M<-rC						I	E	D		
Branch X							I		E	
No op									I	E

Three-stage pipelined timing

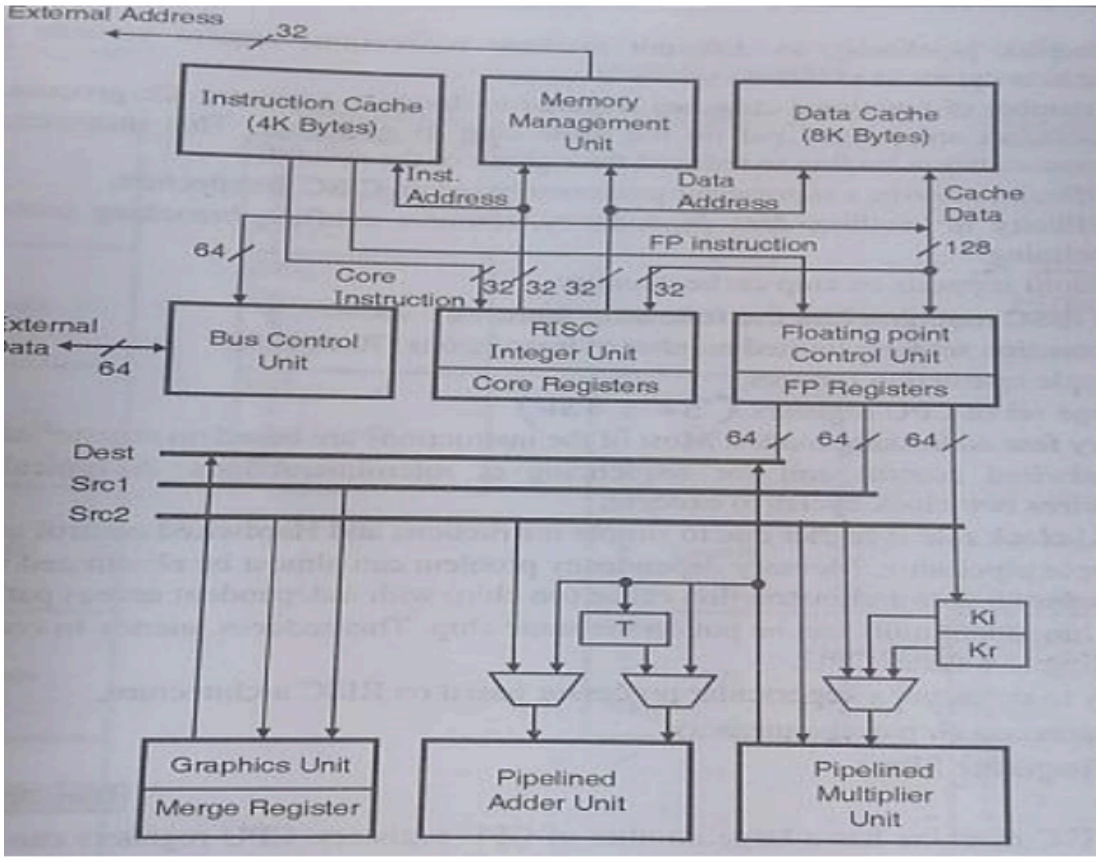
When a cache memory is added, we have two cache memories: data cache and code/ instruction cache. This allows for simultaneous processing of I, E, and D

We have only 8 cycles now

Load rA<- M	I	E	D					
Load rB<- M		I	E	D				
No op			I	E				
Add rC<- rA+rB				I	E	D		
Store M<-rC					I	E	D	
Branch X						I	E	
No op							I	E

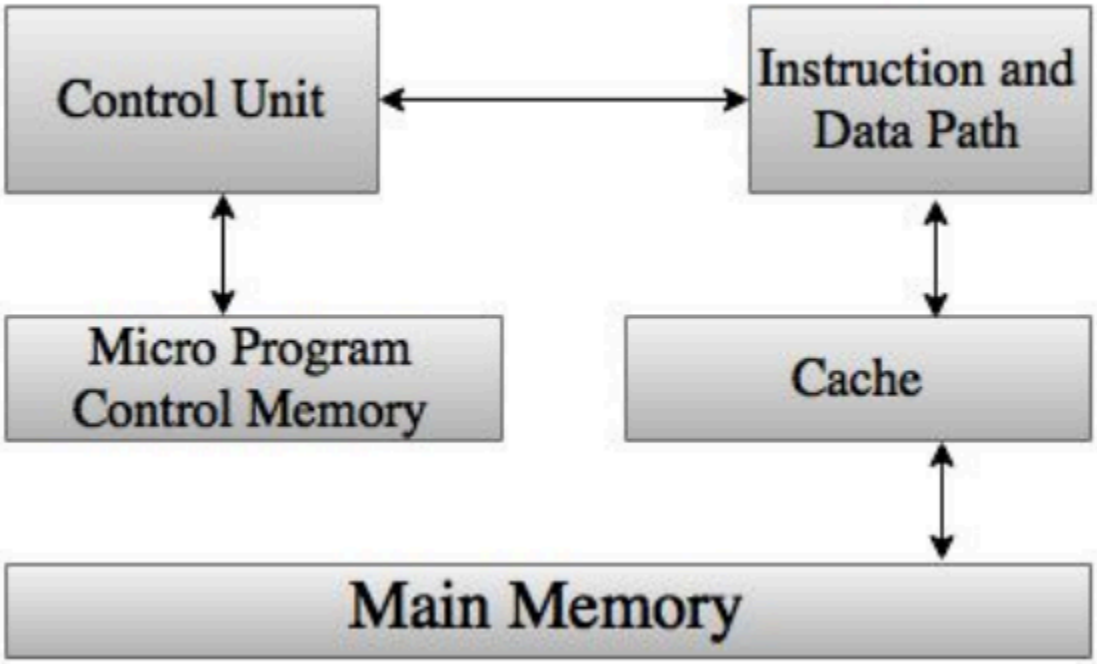
Self learning topics -> skip

Risc Architecture



- Has 9functional units
- All buses are 32 bits wide
- Separate instruction and data cache bus
- The MMU (Memory Management unit) implements paged virtual memory structure.
  Explanation: MMU uses pages to map memory. It breaks memory into small equal blocks. Virtual pages map to physical frames. Page table stores these links.
- The RISC integer unit performs the fetch load etc functions
- It has two floating point units, a multiplier unit, and an adder unit
- It also has a graphics unit that supports 3d drawing

CISC Architecture



//only diag and what each thing does

- Micro Program Control Memory stores instructions that run the CPU.
- Instruction path moves the opcode bits to control unit.
- Data path moves values through ALU and registers.

RISC is hardwired while CISC is microprogrammed.

End of module

Module 4: memory

Memory

Characteristics

1. Location
    a. Memory is located in the CPU, and internally or externally in the computer
2. Capacity
    a. Is measured in two forms
    b. Bytes or Words
    c. Units of memory: byte -> kb -> mb -> gb -> tb -> pb -> eb -> zb ->yb <should know fullform and conversion factor(1024 btw)>
3. Unit of transfer
    a. Internal: by bus
    b. External: by blocks
    c. Addressable unit
       Smallest location which can be uniquely addressed
4. Access Methods
    a. Sequential: Start at the beginning and read in order
       Access time depends on location and prev location
    b. Direct:
       Individual blocks have a unique address
       Jumping to vicinity+ sequential search access
       Time depends on location and prev location
    c. Random
       Individual addresses hold exact locations
       Access time is independent of location/prev location
       Example: RAM
    d. Associative
       Data is located by comparison of contents
       Access time is independent.
       Cache is an example
5. Performance
    a. Access time: Time between request for an operation and its execution is called access time
    b. Memory cycle time: Minimum time elapsed between two consecutive read requests
    c. Transfer rate: Rate of transfer of data
6. Physical Types
    a. Decay
    b. Volatility

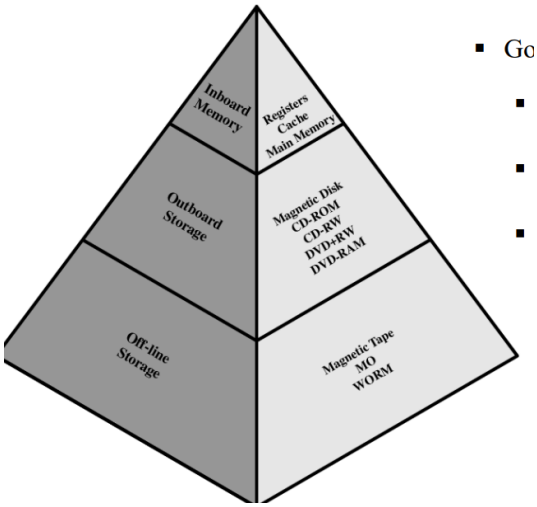
c. Erasability
d. Power consumption

7. Organization
    a. It is the physical arrangement of bits into words
    b. It's not always obvious
    c. To increase the speed of a 32bit processor for example is arranged in 4*8 bits system. In one clock cycle, it can fetch data from four memory bands simultaneously. If it was arranged continually(32*1), it would only be able to access memory from 1 band at a time.

Memory Hierarchy

Registers in CPU
Internal/Main Memory (Cache or RAM)
External memory: disks and drives

Down the hierarchy, cost reduces, capacity increases, and access time increases.2



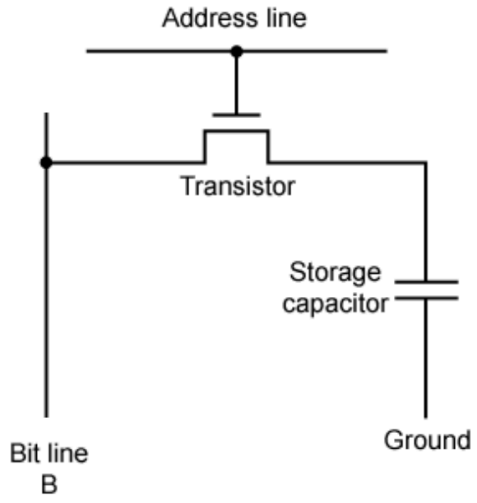
RAM

- Random access
- Read/write
- Volatile: Contents will be gone if the power fails.
- Is a temporary storage
- Static or dynamic memory.

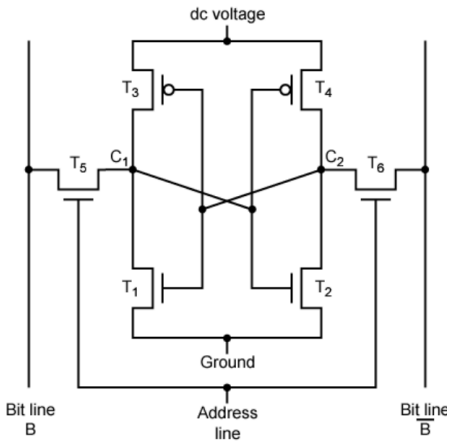
Dynamic RAM Structure: DRAM

1. Bits are stored as charge on a capacitor
2. Charges leak and get lost continuously
3. Need refreshing even when powered
4. Simple construction
5. Less expensive
6. Needs refresh circuits to restore the leaking charge
7. Slower
8. Main memory
9. Essentially this is analog

DRAM stores bits as leaking capacitor charges that need periodic refresh.



Static RAM Structure



When voltage is applied to address line, T5 and T6 act as switches

1. Bits are stored as on and off switches
2. No charges to leak
3. No refreshing is needed
4. Construction is slightly more complicated
5. Larger information is stored per bit
6. It is more expensive
7. Does not need any refresh circuits
8. Is faster
9. Type: Cache
10. This is essentially digital and uses flip flops

Operation

- Transistor arrangement gives a stable logic state.
- State 1:
  c1= high
  c2=low
  t1,t4=off and t2,t3=on
- State 0
  c2=high
  c1=low
  t2,t3=off and t1,t4=on

SRAM vs DRAM

1. Both are volatile-> memory is lost if system fails/powers off
2. Dynamic is simpler, less expensive, needs refresh, and supports larger memory units
3. Static is faster, and is used in cache

Feature	DRAM	SRAM
Storage	Charge on capacitor	Flip-flops (transistors)
Nature	Analog	Digital
Leakage	Charge leaks	No leakage
Refresh	**Required**	**Not required**
Speed	Slow	Fast
Cost	Cheap	Expensive
Circuit	Simple	More complex
Bit density	High (more bits per chip)	Lower
Use	**Main memory**	**Cache memory**

Read only memory (ROM)

- This is permanent storage, non volatile memory
- You can read ROM but cannot write data to it

Types of ROM

1. PROM
2. EPROM
3. EEPROM

PROM: Programmable ROM

- This is written at manufacture time
- Can be programmed ONCE
- Programming happens once using electrical pulses during making.
- Only able to store small amounts of data
- It is less expensive

Erasable Programmable ROM EPROM

- Program, erase, reprogram
- Can be erased by UV
- Can be altered multiple times
- Holds data without power forever

Electrically Erasable Programmable ROM: EEPROM

- EEPROM is a rewritable ROM type, so you can write, but very slowly.
- Is electronically erased
- Flash: Flash is a fast EEPROM that erases blocks together for quicker writes. It can be reprogrammed while still in the computer.
- Can be written anytime without erasing contents
- Less dense and more expensive than EPROM

ROM cartridge

- Used in games
- Prevents software from being copied

Cache

What?

Small amt of fast memory

Is located on CPU chip

Operation

1. Cache fetches data from addresses in main memory
2. CPU checks if the next instruction it needs is in cache
    a. Yes -> Instruction is fetched from cache, very fast
    b. No -> CPU has to fetch from main memory via cache, v slow
3. Cache requires tags to identify which block of main memory is in each cache slot

Tag: A unique identifier for a group of data. Used to differentiate between different blocks of data.

Types of cache

CPU cache is divided into levels. This hierarchy is on the basis of speed and size of cache.

1. L1 (Level 1) cache
   Fast, small, and embedded in the processor chip
2. L2:
   More capacity than L1, located on CPU or a separate chip
3. L3:
   Used to increase the performance of L1 and L2. It is slower than them both but has double the speed of the RAM.

Cache Design

Size

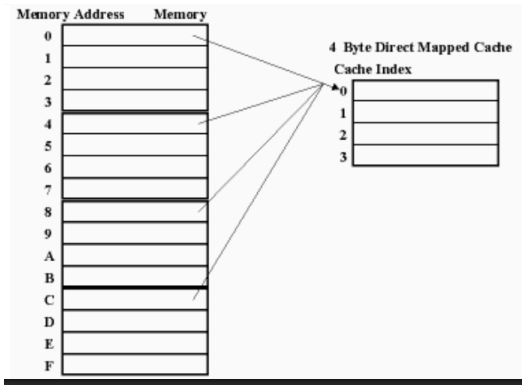
Bigger size -> expensive and faster but after a point, it takes more time to check for data

Cache Mapping technique

It is the rule that decides which cache line will store a given main-memory block.

1. Direct

   Contiguous blocks are mapped onto the ache byte-size wise.
   Direct mapping rule:
   Cache line = (Memory block number) mod (number of cache lines)



Example:
If the cache has x blocks of y words, it is of x*y size.
Total address size in 16 bit:

Tag	Block/Line(128)	Word(16)
5(16-(7+4))	7(2^7)	4 (2^4)

Cache line 0 can hold block 0, m, 2m, etc
Cache line 1 holds 1, m+1, 2m+1, etc
Cache line m-1 holds m-1, 2m-1... and so on
In direct mapping, each memory block only has ONE possible cache line, decided by the formula.

Working

1. Word (a very small unit data) field selects one word from the 16 available
2. Line field selects the line in the cache where this word will go
3. The tag field checks if the correct word is in the correct field/line. (hit)

Pros and cons

1. Simple
2. Cheap
3. Con: Fixed location. More chances to have cache misses as two blocks can be mapped to the same line.

Formulae:
WORD bits = $\log_2$(block size in words)

INDEX bits = $\log_2$(number of cache lines)

TAG bits = total address bits − INDEX − WORD

Format:
TAG | INDEX | WORD

2. Associative
    a. Fully associative
    Memory blocks can go into ANY cache line: no fixed formula or rule.
    Working:
        - The tag field uniquely identifies the memory block stored in a cache line, because any block can go to any line.
        - All cache tags are searched to find if the tag matches any
        - If so, we have a hit.
        - If not, there's a miss.

- Miss -> Replace one of the cache lines by this tag field identified line before reading/writing into the cache.

Formulae:
WORD bits = $\log_2$(block size in words)

INDEX bits = 0
(because any block can go anywhere)

Format:
TAG | WORD

TAG bits = total address bits − WORD

b. Two Way Set associative
   Cache is 128 blocks of 16 words.
   Set is divided into 2: 64*16
   Main memory -> 4k blocks * 16

   Working:
   - Memory blocks are divided into sets.
   - Tag field identifies the memory block within the selected set.
   - Each set has room for only 2 lines at one moment. So the search for the match is limited to those two lines. (Contrary to fully associative, where the entire cache is searched)
   - Hit -> read and write process can begin
   - Miss -> Replace one of the two cache lines by this line before proceeding
   - In set-associative mapping, when the number of lines per set is n, it is called n-way set associative. Hence this is 2-way associative

Pros and cons:
1. Main memory can load into any line of cache
2. Expensive cache searching

Formulae:
WORD bits = $\log_2$(block size in words)

Number of sets = (number of cache lines) / n
n-> associativity (2 way set associative, n=2)

SET bits = $\log_2$(number of sets)

TAG bits = total address bits − SET − WORD

Format:
TAG | SET | WORD

Cache coherence

Each CPU has its own cache in multiprocessor systems. The same data might thus appear in multiple caches. If one CPU changes the data, the other caches might hold the older versions. This is inconsistent and undesirable. This is the cache coherence problem.

Solutions

1. **Software**
   Software solutions to cache coherence problems are handled by the compiler and operating systems.
   - The compiler flags the variables that are subject to change often.
   - The operating system stop these variables from being cached
   - Responsibility of maintaining coherence moves from hardware to software
   - Although this is simpler hardware implementation, it takes much more time
   - Not used very often as it is not efficient at runtime.
2. **Hardware**
   Hardware solutions are handled by cache controllers that follow certain protocol.
   - It detects conflicts dynamically at runtime
   - The process is automatic and transparent to programmer
   - Caches are in communication to keep data consistent.
   - Cache is more effectively used.
   - Commonly used in multiprocessors.
   - Called **snoopy protocol**

     SNOOPY PROTOCOL
     - Each cache snoops the bus (monitors the bus)
     - When one CPU updates a block, other CPUs are informed
     - Cache recognises when a block is shared
     - No coherence problem.

- This is suited for bus-based multiprocessor systems but it can increase bus traffic.

Write policies

Write through

1. Every write goes to cache and main memory simultaneously.
2. Other CPUs watch the memory bus to update their caches.
3. Simple, but causes heavy traffic and slower writes.

Write back

1. Writes only go to the cache first
2. An update bit called the dirty bit(dirty bit = block that causes incoherence) is set
3. Data goes to the main memory only after the dirty bit is evicted.
4. Faster, but caches might go out of sync (cause incoherence)
   This happens because while all this is happening and the dirty bit is being evicted older CPUs have the older data which is different from this evicted data.
5. Important to manage I/O carefully to see correct data.

Write-Update

1. If a CPU updates a word, the new value is broadcast to all caches
2. Useful when many CPUs can read and write on the same block
3. It causes more traffic than write-back

Write-invalidate

1. Before writing, CPU invalidates this block (which has copies in other CPUs) in all other caches.
2. CPU now has exclusive access to modify the block
3. Other CPUs will have to reload to access
4. Works for multiple reader one writer situation
5. Leads to MESI protocol

MESI Protocol

Each of the cache line is in one of the four states:

1. M-> Modified
   Cache has updated the data. Main is outdated.

Only THIS cpu has the updated data, and this block must be written back to memory before another CPU can use it.

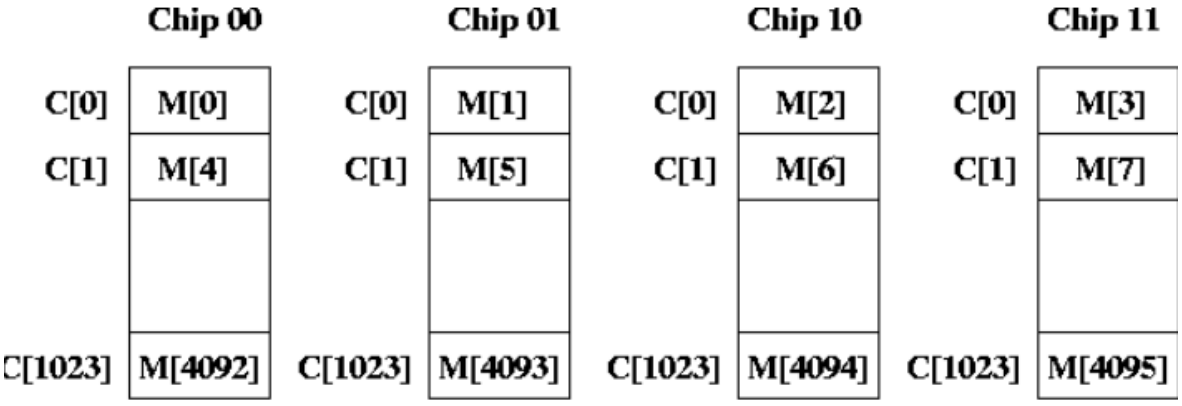
2. E-> Exclusive
   Only cache has the clean copy. The memory is updated.
   If the CPU writes to it, the state of the cache line becomes Modified and if a CPU reads from it, the state of the cache line becomes shared.
3. S-> Shared
   Many caches share the same data, and all copies are clean.
   If a CPU wants to write, it must first invalidate other CPUs.
4. I-> Invalid
   This cache line is invalid. Perhaps another CPU has updated it. User must fetch a fresh copy before using this cache line.

These states ensure correct behavior during reads, writes, sharing, and invalidation.

Interleaved and Associative Memory

Interleaved memory

- A design to compensate the slow speed of DRAM (Dynamic RAM)
- Instead of putting all consecutive addresses in one memory module, the addresses are spread evenly across multiple modules.
- This means when the CPU reads or writes a long block of data, different modules can work at the same time, so memory operations overlap.
- This results in higher memory throughputs due to less writing. The CPU waits for less time and more data is transferred parallely.



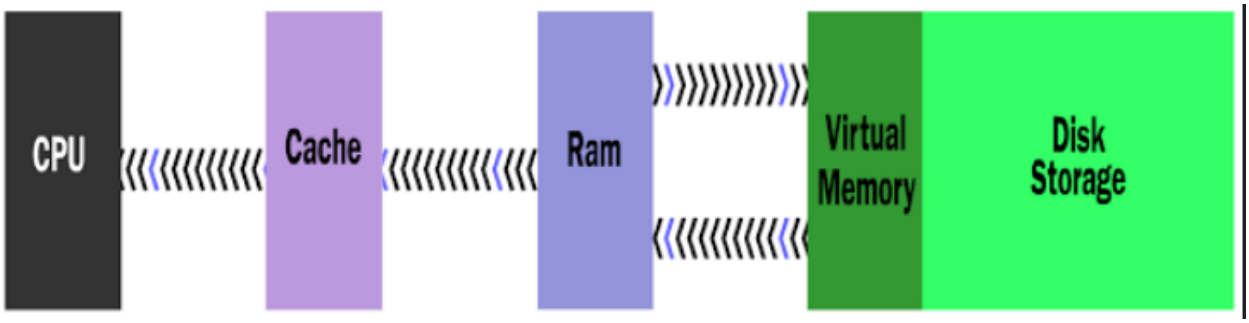
Associative memory

- This type of memory is searched by content instead of a numeric address.

- Normally we give addresses, but here we give a pattern or a clue and the memory checks all its entries to find the match in parallel.
- The search key is associated with stored data, hence associative memory

Virtual memory

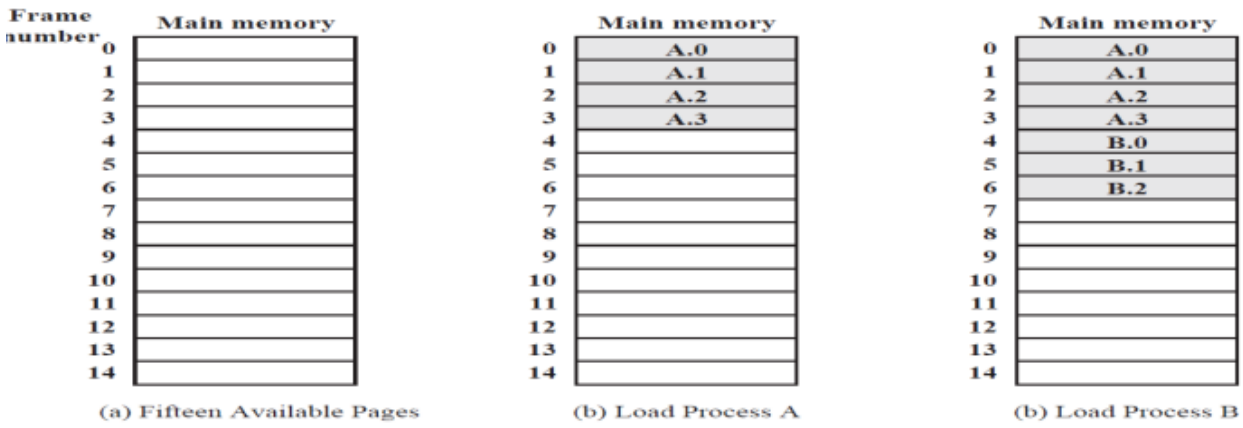
- A technique where the OS uses disk space as extra RAM.
- Lets programs think they have a large, continuous memory
- Allows more programs to be opened at the same time (Spotify in the background, web browser, word document, whatsapp notifications)
- Uses page tables, TLBs, and address translation.
- Much larger but slower.



It frees up the RAM space to load the new application.
Copying happens automatically. Since the space on hard disk is cheaper, virtual memory has an economic benefit.
The operating system has to keep swapping information between the RAM and harddisk, this leads to thrashing. Thrashing: computer feeling slow.
We can convert the virtual address into physical main memory by paging and segmentation.

Paging

- Paging is a memory-management scheme
- Here, the main memory (hardware) is split into small, equal-sized chunks called page frames.
- Each process (software) is split into blocks of the same size. These blocks are called pages.
- Because both sides (main memory and process) use equal sizes, pages can fit into any free page frame, avoiding the inefficiency of old variable-size partitions.
- Pages that were recently used stay in memory, while others may be moved out.
- A page table keeps track of which page of a process is stored in which frame, allowing the OS to manage memory efficiently and avoid external fragmentation.



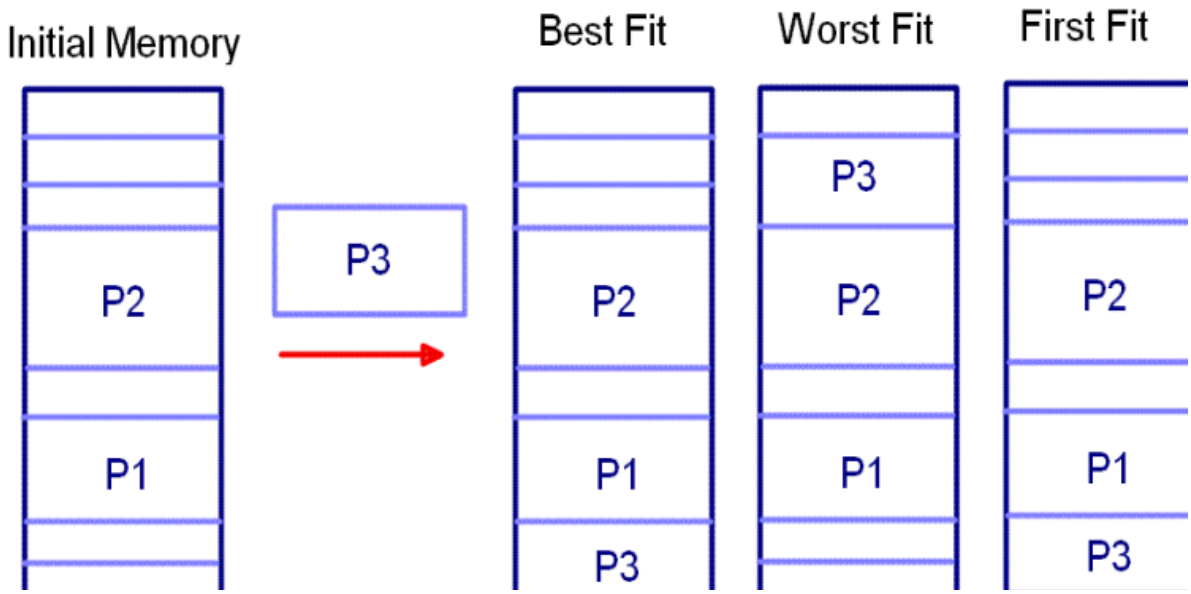
(a) Fifteen Available Pages (b) Load Process A (b) Load Process B

Segmentation

- Paging limitation - it leads to internal fragmentation.
- Segmentation divides a program into logical parts called segments: functions, arrays, stacks, code modules, and data structures. Each segment has a different size, so we have “variable sized partitions” to accommodate these partitions.
- This is contrary to what happens in paging where there is constant sizing for each component.
- Contiguous memory blocks / segments of a process are not loaded at the same time.
- Like a page table, here a segment table is required.

Main memory allocation

- Memory is divided into set of contiguous locations called segments / regions / pages
- Placement of blocks of information in the memory is called memory allocation
- Memory management systems keep the information in a table that contains all the available free slots.



The first fit is when you scan memory from the top, which is the first space big enough to hold your memory
The best fit is one where when you place memory, you minimize the leftover /unusable space (best fit -> try to find closest to exact size match)
Worst fit is where you place the memory poorly, choosing the largest hole available, leaving more unused space and not utilizing the better possibility of space for a memory. Notice in the diagram how placing P3 above P2 created unused space. Comparing best and worst, best can store sizes 1,2,3 now but worst can only store size 1,2 now.

Replacement algorithm

When the cache is full and a new block must be loaded, the system must decide which existing block to remove. This decision is made by a replacement algorithm, usually implemented in hardware for speed.
Types:
LRU: least recently used
FIFO: first in first out
Optimal
RANDOM: choose any random block to remove
//numericals

Secondary Storage

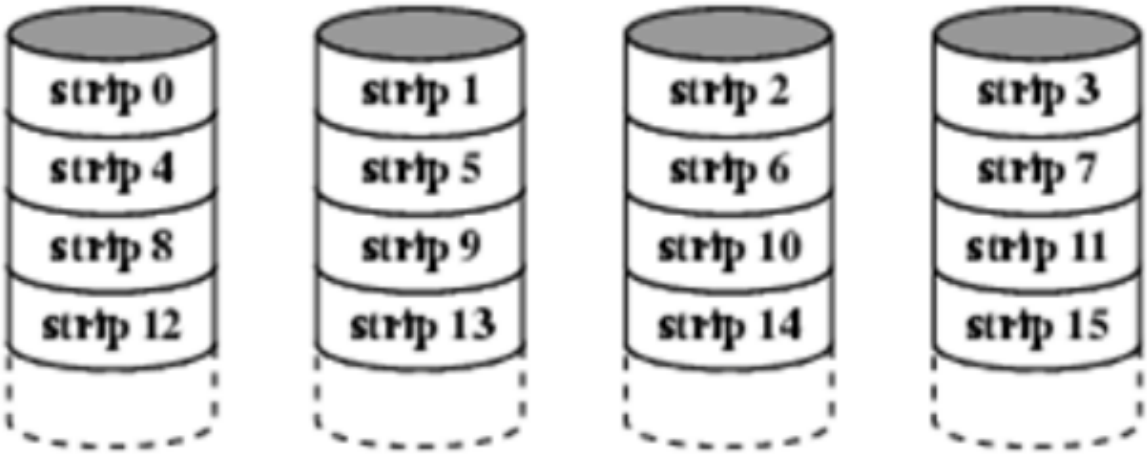
Anything externally attached to computer for storage: magnetic disk, floppy disk, RAID, DVD, CD, optical memory

RAID: REDUNDANT ARRAY OF INDEPENDENT DISKS

For complex redundancy and performance, we need RAID to store data effectively. There are 0 to 6 levels of RAID.

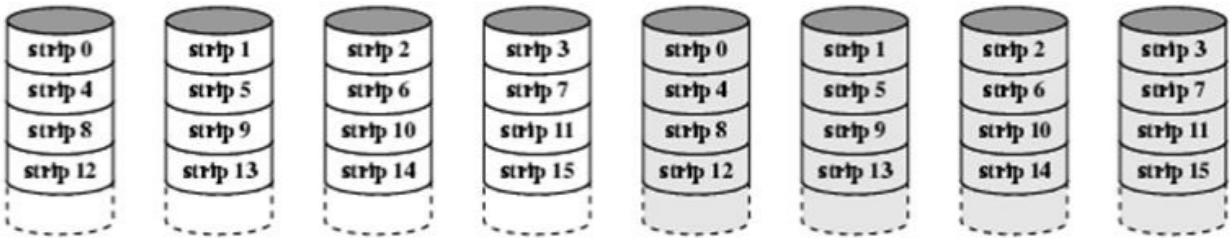
Level 0:

- Non redundant
- RAID 0 uses striping, which splits data across multiple disks to increase speed. It offers high performance but no redundancy, so if one disk fails, all data is lost.



Level 1

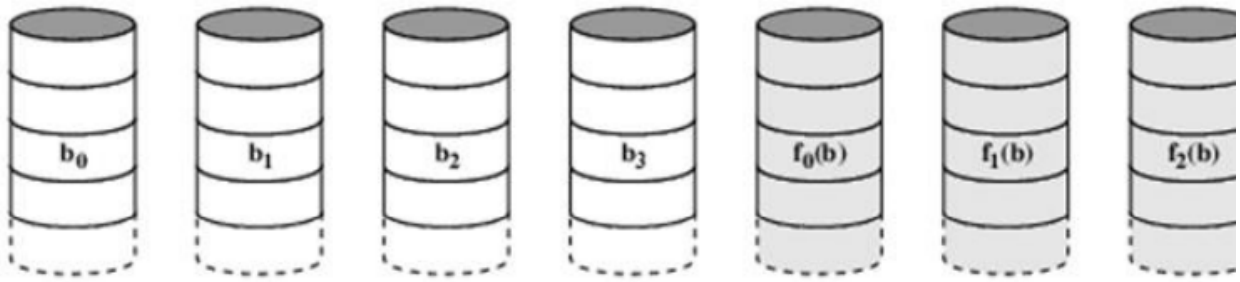
- Mirrored
- It stores data across multiple strips but also mirrors them. Just in case a disk fails and its data is lost, you have its mirror to retrieve.
- Giving high reliability but no speed gain and double storage cost.



Level 2

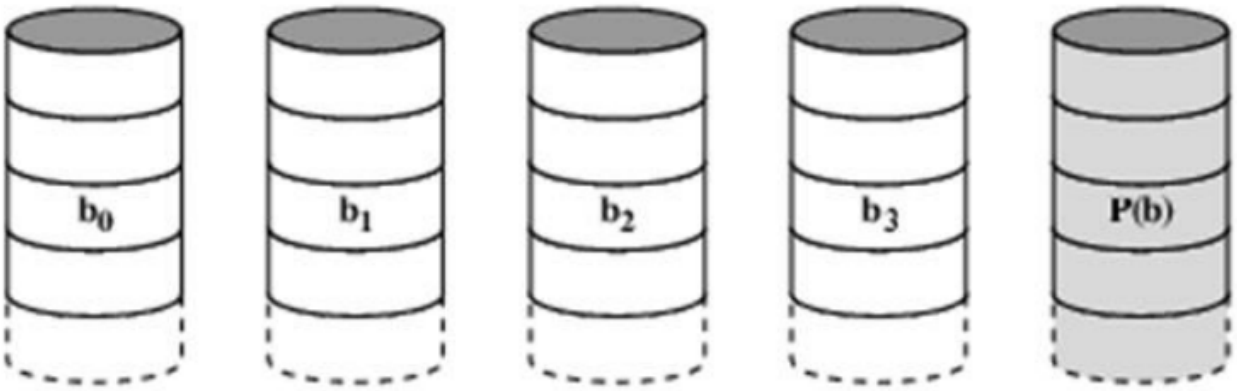
- Hamming code

- Bit level striping with dedicated hamming code. Data is split at the bit level.
- Strong error correction
- Grey boxes: disks for hamming code ecc bits/ parity disks
- Rarely used since it needs a lot of hardware and simpler RAIDs 0 and 1 are more efficient.



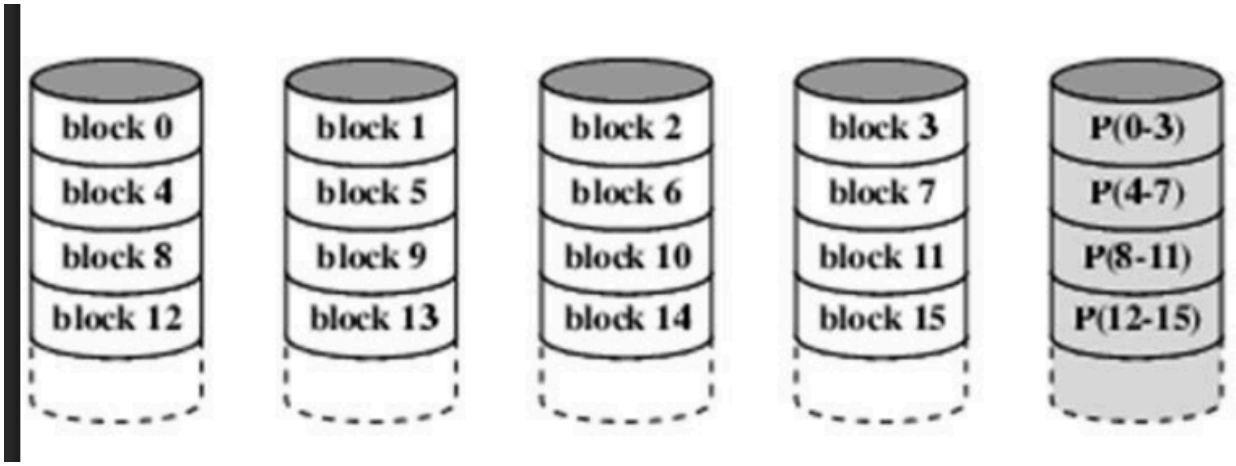
Level 3

- Bit interleaved parity
- Splits the data at the bit level and uses one dedicated parity disk (grey) for error detection and recovery
- All disks are in sync, allowing faster data transfer
- Performance drops for many independent small requests.



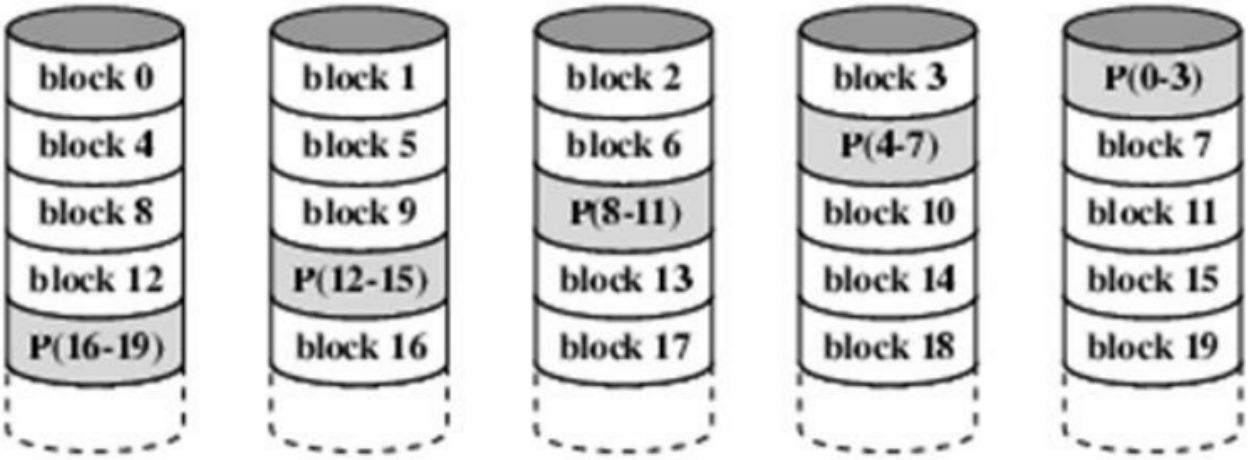
Level 4

- Block level parity
- Has one dedicated parity disk
- Each block goes to a different disk, the parity disk stores parity for each block set.
- Singly parity disk becomes bottleneck as every write operation has to update it.



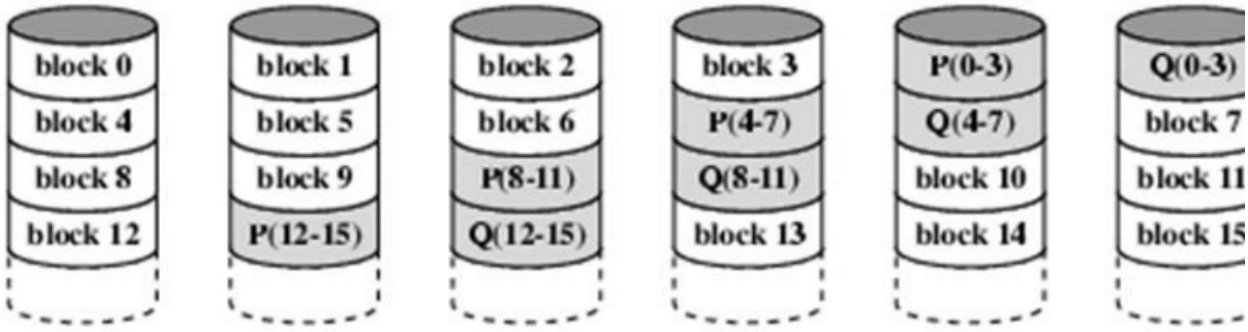
Level 5

- Block level distributed parity
- Parity disks distributed across
- Solves bottleneck problem
- Better write performance, fault tolerance, good efficiency



Level 6

- Dual redundancy
- Double distributed parity
- Across all disks, it stores two different parity values allowing the system to survive two disk failures.
- Very high reliability, slow write time as two parity values need to be updated per write.



Formulae

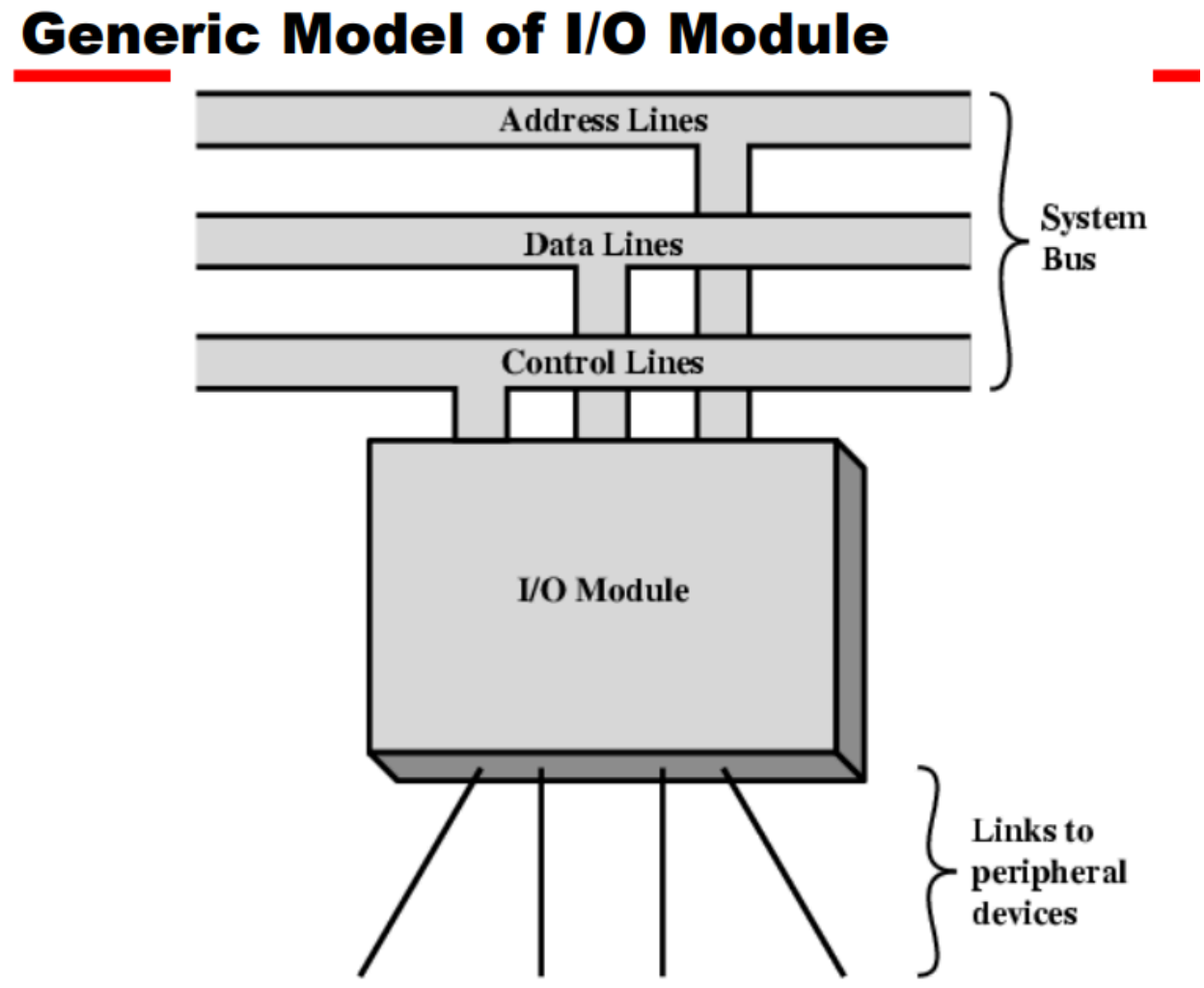
1. Size of main memory = no of blocks * size of each block
2. Tag size = no of blocks in main memory / no of sets in cache
3. No of lines in each cache set = cache size / no of ways
4. Set size = log2 (no of words in block)

Module 5: Input/output

Input/Output

We have a variety of peripheral devices to perform input and output. Each has a different amount of data to deliver, at their own speed, in their own formats.
All peripherals are slower than the CPU and the RAM. so we need I/O modules.

The I/O module serves as a bridge between the CPU–memory system and the external devices, allowing data to flow between the computer's internal components and the peripherals.



Functions of IO module

Cp ded

1. Control and timing: coordinates operations

2. Processor communication: sends status commands and data between CPU and peripherals
3. Device communication
4. Data buffering: Temporarily stores data to match the speed difference between fast CPU/memory and slow devices.
5. Error detection: Checks for transmission errors (parity, checksums) and reports problems to ensure correct data transfer.

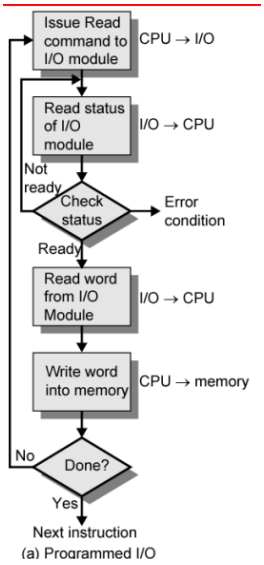
The external devices like screen printer keyboard are readable by the humans. The internal devices like monitoring and control systems are machine readable. Communication between these happens thru communication devices like the modem or network interface card (NIC)

I/O steps

1. CPU checks I/O module device status
2. I/O module returns status
   If ready-> CPU requests data transfer
3. I/O gets data and transfers it to CPU

I/O techniques

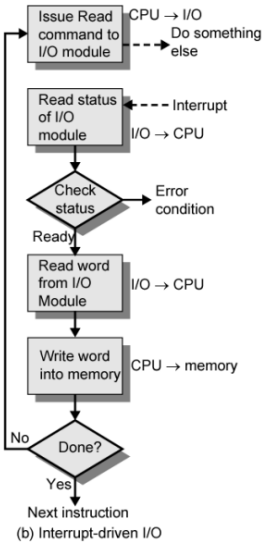
Programmed I/O



Every step of the input output transfer is handled by the CPU. CPU runs instructions that initiate the request of I/O, direct the data movement to or from the I/O and terminate the operation after being done. Since everything is done by the CPU, it keeps checking the device status and cannot do anything else simultaneously.
This is useful when hardware costs need to be minimized as there's no extra hardware needed to initiate programmed I/O
Steps:

1. Read i/o status bit
2. Test status bit to check if peripheral device is ready
3. If not ready, CPU wastes its time till device becomes ready
   If ready, read word from I/O module (I/O -> CPU) and write word into memory. (CPU -> memory)

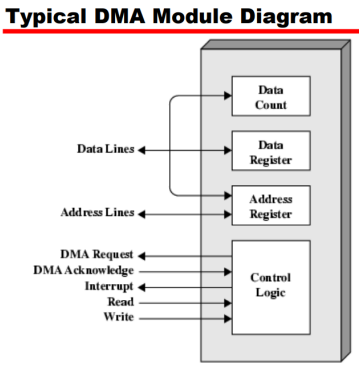
Interrupt Driven I/O



Major drawback of programmed I/O is busy waiting: when the CPU wastes its time waiting for the peripherals to become ready. This happens often as the I/O speed is less than CPU. the CPU performance goes down as it has to continuously check for peripheral devices' status. The solution to this problem is the interrupt mechanism.

This implies that while a device is not ready, the CPU should switch to another while this one becomes ready. Interrupt driven i/o overcomes cpu waiting, eliminates the repeated checking of CPU for the devices and the I/O module interrupts the running process as soon as it becomes ready.

Direct memory access



Drawback of the interrupt driven and programmed I/O is that they both require active CPU interference. Thus, the CPU is tied up and the rate of transfer is bounded.
DMA or direct memory access allows some hardware to access the RAM independently, without ever talking to the CPU.

CPU tells the DMA controller the device address, starting memory, amount of transferable data, and the operation: read or write. Then, the CPU goes on to do its work and the DMA controller deals with whatever transfers need to be made. It sends an interrupt to the CPU when it's done working. In a gist, CPU is DMA's manager and DMA is an assistant who will do everything independently as asked and just let the manager know once it's done.

DMA transfer modes

1. Burst Mode
   A block of data of some arbitrary length is transferred in one single "burst".
   A burst is a temporary, high-speed data transfer made to facilitate sequential data transfer at max throughput (speed).
2. Cycle stealing mode
   DMA controller can use a system bus to transfer **one word at a time.**
   After this, the control of the system bus must return to the CPU.

The DMA uses the system bus **only if the processor does not need it.** Or it must force the processor to stop using the system bus temporarily. Is called cycle stealing because DMA steals a bus cycle from the CPU.

3. Transparent mode
   DMA uses only the idle CPU memory cycles, taking one cycle at a time when the CPU isn't using the bus. This implies that forcing the CPU to stop using the system bus is not allowed. So in essence, the CPU never notices or slows down. Transfers run entirely in the background, making DMA invisible (transparent) to the CPU.

Advantages

1. High transfer rate
   Fewer CPU cycles for each transfer
   DMA speeds up by minimizing CPU involvement in processes
2. Work overload on the CPU reduces.

Disadvantages

1. DMA needs a DMA controller
2. Is more expensive
3. To avoid the cache coherence problem, some synchronization mechanism must be provided.

—----------end—----------

//refer to the all diagrams tab for diagrams

Module 6: Multiprocessor Configuration

Parallel Processing

Parallel processing increases performance by doing multiple operations parallely.
This is done through parallelism, like using multiple ALUs inside a single CPU or using multiple processors that work together.

Flynn's Classification

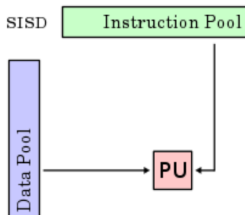
It is based on how instruction streams and data streams (internal organization of processors) are organized, which depends on processor structure and interconnection networks.
Data and processors in parallel system are organized in streams:

1. An instruction stream is the flow of instructions from the memory to the processor.
2. Data stream is the flow of operands going into a processor and results coming out.

These two streams help define parallel computers.

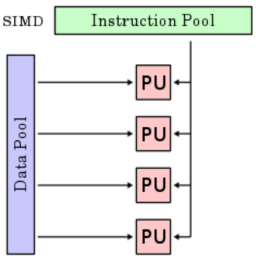
Types

SISD



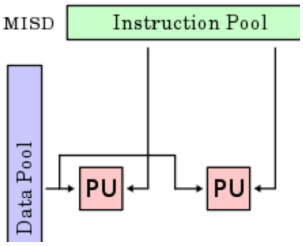
Single instruction stream, single data stream
Has a single processor, called the uni-processor

SIMD



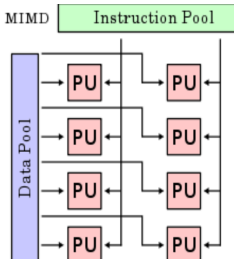
Single instruction stream, multiple data streams
A single instruction controls multiple processing elements at the same time. Each processor has its local data and executes the same instruction on multiple data streams. This is used in array and vector processing for high speed parallel computation.

MISD



Multiple instruction streams, single data stream.
A single data stream is sent to multiple processors. Each runs a different instruction sequence on the same data.
This is purely theoretical and is never practically implemented.

MIMD



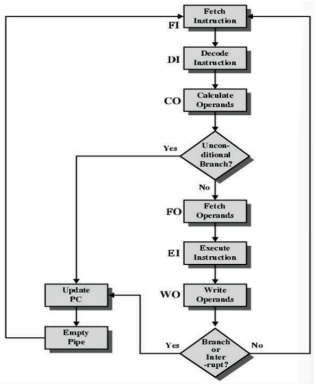
Multiple instruction streams, multiple data streams
Multiple processors operate on multiple data streams at the same time.
Used in real systems like NUMA machines.

Pipeline Processing

Pipelining: Pipelining means doing different stages of a task at the same time so that multiple instructions are in different phases together.
It's the overlapping of instruction execution stages, which increases throughput.
It allows the next instructions to be fetched while the processor is performing some arithmetic operations. Then it holds the fetched instructions in a buffer until each instruction operation can be performed. Instruction fetching is continuous. Throughput is enhanced.

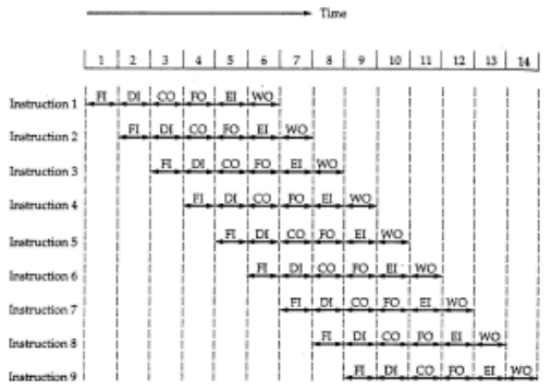
Stages of instruction pipelining

Six Stage Instruction Pipeline



1. Fetch instruction FI
   Read the next expected instruction into the buffer
2. Decode instruction DI
   Determine the opcode(which operation to perform) and the operand specifiers(registers/memory addresses/immediate values).
3. Calculate Operands CO
   Calculate the effective address of each source operand.
4. Fetch operands FO
   Fetches operands from the memory.
5. Execute instruction EI
   Performs the intended operation and stores result
6. Write Operand WO
   Stores the result in memory

Timing diagram for 6 stage instruction pipeline



Pipeline hazards

What?
Any cause that stalls the pipeline is called a hazard.
Like resource usage (it’s dependent on inter-instructions) and job scheduling problems.
Three types:

1. **Resource conflict**
   When the resources are insufficient. When two pipeline stages need the same hardware resource at the same time. Example: both instruction and data streams need the same memory.
   This causes structural hazard.
2. **Data/data dependency conflict**
   When the new instruction depends on a previous instruction’s result and the previous instruction hasn't completed yet.
   RAW, WAR, WAW are the data hazards. RAW means read after write, WAR = write after read and WAW = write after write. RAR is not a read hazard as it does not interfere with each other.
   Solution: use pipeline scheduling.
3. **Branch difficulty (PC contents got changed)**
   Happens because the pipeline does not know which instruction to fetch.
   Solution:
   a. **Flush pipeline:** Cancel the wrong instructions after branch decision
   b. **Delayed branching:** Execute an extra instruction after the branch
   c. **Conditional branching:** Wait the pipeline until the branch condition is evaluated.

Design Issues of Pipeline Architecture

Pipeline design issues focus on how instructions flow and how arithmetic units operate.

1. Instruction pipeline design:
   - In-order issuing: Instructions are fetched and issued in the original order.
   - Out of order issuing: Instructions issue as soon as their operands are ready.
   - Reorder issuing: Hardware reorders instructions to maintain correct results.
2. Arithmetic pipeline design:
   - Deals with arithmetic operations ka pipelining
   - For each arithmetic operation (floating point, integer, fixed point), a different pipeline is set aside. Each has a different timing and stage.

Principles of Designing a Pipelined Processor

- Proper data buffering avoids congestion and makes pipeline operations smooth.
- Understand instruction dependencies to prevent data hazards.
- Detects and resolves logic hazards (data, control, structural).
- Avoid collisions by making operations sequential.
- Avoid structural hazards by providing enough hardware.
- Allow pipeline reconfiguration.

Instruction Set

Data transfer instructions

General purpose data transfer group

These instructions move data between registers and memory.

- MOV – Copy data from source to destination.
  Source and destination for MOV:
  - Accumulator to memory (and vice versa)
  - Register to register
  - Register to memory (and vice versa)
  - Register to immediate
  - Memory to immediate

  Eg: MOV AX,BX
- PUSH – Place data on the stack.
  SP = SP-2
  SS:[SP] = operand
  Decrements the stack pointer by 2 and copies a word from src to destination. The operand source can be a register or memory
  The stack ptr and stack segment register must be initialized before using.
  PUSH is used to save data on the stack so it's not destroyed by the execution of successive instructions.
- POP – Remove data from the stack into a register or memory.
  Operand=SS:[SP](top of stack) SP=SP+2
  Copies a word from the stack segment to a destination specified in the instruction.
  The destination can be a register or the memory.
  Data is not changed.
  Stack pointer is incremented by 2 after this operation.
- XCHG – Exchange data between two registers or between a register and memory.
  Register can be 8-16 bits
  Cannot directly exchange content between two memory locations.
  The src/dest must either both be words or both be bytes.
- XLAT – Translate a byte using a lookup table
  AL=DS:[BX + unsigned AL]
  Copies the value of memory byte at location DS"[BX+unsigned AL] to AL.
  Used to convert BCD (binary coded decimal) to 7 segment code (as on a digital display).

Input and output data transfer group

- IN - Read data from an IO port to AL or AX
  Two formats:
  - Fixed port: port address is written in instruction
    IN AL, C8h
  - Variable port: port address is stored in DX
    MOV DX, 0FF0h
    IN AL, DX
- OUT - Send data from AL/AX to IO port
  Two formats:
  - Fixed port: 8 bit port address is given
    OUT 3Bh, AL
  - Variable port: DX holds the port address
    MOV DX, FFF8h
    OUT DX, AL

Address group

- LEA - Calculates the offset address of the memory variable given as the source. Places this offset value into the 16-bit register.
  Example: LEA AX, COUNT
- LDS- Loads the offset from memory into register and segment into DS.
- LES- Loads pointer with ES (Extra segment).
  LES loads a 32-bit pointer from memory: the offset goes into a register and the segment goes into ES.
  It is used to access data stored in an extra memory segment.

Flag transfer group

These instructions are related to movement of flag register to/from a register and memory.
AH is the upper 8 bits of the AX register in the 8086 processor.

- LAHF: Load AH from flags
  Copies the lower byte of the FLAGS register into AH.
  Lets programs read flag states (ZF, SF, PF, CF, AF).
  Undefined bits stay unpredictable.
- SAHF: Store AH into flags
  Copies the contents of AH back into the lower byte of FLAGS.
  Used to restore or modify key flag bits.
  Undefined bits remain unchanged or unpredictable.
- PUSHF: Push Flags to Stack
  Decrements SP by 2 and pushes the entire FLAGS word onto the stack.
  Useful for saving processor status during procedures or interrupts.

- POPF: Pop flags from stack
  Pops a word from the top of the stack into the FLAGS register.
  Restores previously saved flag states and increments SP by 2.

Arithmetic group: Addition

- ADD
  Add byte or word
  Format: ADD dest, src
  Adds a number from src to number from dest and puts result in dest
  ADD AX, BX → AX = AX + BX
  Some source/dest pairs are
  reg reg, reg mem, mem reg, reg immediate, mem immediate, accumulator immediate
- ADC: add with carry
  ADC AX,BX
  Adds the src, dest, and carry flag and stores result in dest.
- INC: increment byte or word by one
  INC AX
  Format: INC dest
- AAA: Ascii adjust after addition
  AAA fixes the result after adding two ASCII decimal digits.
  ASCII codes for digits don't add cleanly, so AAA checks AL; if the lower part is above 9 or the adjust flag is set, it adds 6 to AL and adds 1 to AH for the carry. This makes the result a proper unpacked BCD digit.
- DAA
  DAA fixes the result after adding two packed BCD bytes (two decimal digits). If the lower 4 bits of AL are above 9 or AF is set, it adds 06h. If the upper part is too large or CF is set, it adds 60h. This makes AL a proper packed BCD value.

Gist of aaa and daa: AAA fixes the result after adding ASCII digits so AL becomes a correct decimal digit. DAA fixes the result after adding packed BCD digits so AL becomes a proper BCD value.

Arithmetic group: Subtraction

- SUB: same as ADD but for subtraction
- SBB: subtract with borrow (This instruction subtracts source and carry flag(i.e. Borrow) from destination.)
- DEC: opposite of INC
- NEG: negate the byte
  Format: NEG dest, example: **NEG AX**

- CMP: compare byte or word
  CMP dest, src
  Eg: **CMP BH,CL**
  This instruction compares a word/byte from source with byte/word from destination. The comparison is done by subtracting the source byte or word from the destination byte or word.
  The result is not stored, only the flags get updated. The src might be reg/immediate/memory and the dest has to be either reg or memory
- AAS: same as AAA for subtraction: adjust ascii after subtraction
- DAS: same as DAA for subtraction. Adjust decimal after subtraction.

Arithmetic group: Multiplication

- MUL
  Format: MUL src
  MUL AX
  Multiplies an unsigned bit from src with unsigned bit in AL register
  OR multiplies an unsigned word from src with an unsigned word in AX
  Result in AX for byte, in DX and AX for word
- IMUL
  Format: IMUL src
  IMUL does signed multiplication.
  Byte × AL gives result in AX.
  Word × AX gives the result in DX:AX.
  Extra bits are sign-extended, and CBW is used to extend signed bytes before multiplying words.
- AAM
  AAM fixes the result after multiplying two unpacked BCD digits.
  It takes the value in AL, divides it by 10, puts the quotient in AH and the remainder in AL, giving a proper two-digit BCD result.

Arithmetic group: Division

AL is the lower 8 bits of AX part of the 8086 processor and AH is the upper 8 bits.

- DIV
  Format: **DIV src**
  DIV performs unsigned division.
  If the source is a byte, AX ÷ source → quotient in AL and remainder in AH.
  If the source is a word, DX:AX ÷ source → quotient in AX and remainder in DX.
- IDIV

IDIV is signed division.
If the source is a byte, AX ÷ source → signed quotient in AL and signed remainder in AH.
If the source is a word, DX:AX ÷ source → signed quotient in AX and signed remainder in DX.

- AAD
  AAD prepares two unpacked BCD digits for division. It converts the digits in AH (tens) and AL (units) into one binary number using: AL = AH*10 + AL, and sets AH to 0. This lets DIV produce a proper BCD quotient in AL and remainder in AH.
- CBW: Convert byte to word
  CBW sign-extends AL into AX.
  If AL is negative (MSB = 1), AH becomes FFh; if AL is positive, AH becomes 00h.
  Used before signed division so AL is properly extended into a 16-bit signed value.
- CWD: Convert word to double word
  CWD sign-extends AX into DX:AX.
  If AX is negative (MSB = 1), DX becomes FFFFh; if AX is positive, DX becomes 0000h. It's required before doing signed word division with IDIV so the dividend is a proper 32-bit signed value.

Bit manipulation instructions

1. NOT: flips the bit from 0 to 1 or 1 to 0
   The destination can be reg or mem but not a segment register or immediate data.
   Format: **NOT dest**
2. AND: applies the AND operation on both
   Format: **AND AL,BL**
   Result is stored at a specific location
   Src can be reg/mem/immediate number but destination can either be reg/memory location
3. OR: applies the OR operation on both
   Format: **OR AL,BL**
   Result is stored at a specific location
   Src can be reg/mem/immediate number but destination can either be reg/memory location. Both together cannot be memory locations.
4. XOR: applies the XOR operation
   XOR AL,BL
   Stores result in destination. Dest can be reg or mem and src can be reg/mem/immediate number. Both src and dest cannot be memory locations.
5. TEST
   TEST dest, src

TEST AL, 75h

This instruction ANDs contents of a source byte or word with contents of specified destination word.
AND performs a bitwise AND and stores the result in the destination, updating flags.
TEST performs a bitwise AND only to set flags (ZF, SF, PF) and does NOT change either operand.

Program for addition,subtraction,multiplication and divion of two numbers

```
data segment
a dw 1101h
b dw 1011h
c db 05h
d db 02h

radd dw ?
rsub dw ?
rmul db ?
rdiv db ?
rrem db ?
data ends

code segment
assume cs:code, ds:data
start:
mov ax,data
mov ds,ax
mov ax,a
mov bx,b
add ax,bx
mov radd,ax

mov ax,a
mov bx,b
sub ax,bx
mov rsub,ax

mov al,c
mov ah,d
mul ah
mov rmul,al

mov al,c
mov bl,d
mov ah,00h
div bl
mov rdiv,al
mov rrem,ah
mov ah,4ch
int 21h
code ends
```

LHS is question and RHS is uska answer

Expect aisa question for “Assembly coding” in syllabus.

mov ax, data : move data to ax

mov ds,ax : move content of ax to ds

mov ax,a: move content of A to ax

mov bx,b : move content of B to BX

add ax,bx: Add ax and bx and store in ax

mov radd,ax: move content of AX to RADD

mov ax,a mov bx,b: move content of a and b to ax and bx

sub ax,bx, subtract bx from ax and store result in ax

mov rsub,ax : store ax content in rsub

mov al,c : move content of c to AL

mov ah,d: move content of D to AH

mul ah: Unsigned multiply: AL × AH → result in AX. (AL = low byte result, AH = high byte result)

mov rmul,al: Store content of AL in RMUL

mov al,c: Move c to AL

mov bl,d: Move d to BL

mov ah,00h: Move content of address 00h to Ah

div bl: Divide BL

mov rdiv,al: Store content of AL in RDIV

mov rrem,ah: Store content of AH in RREM

mov ah,4ch: Load DOS exit function number.

int 21h: Terminate program

Probable Questions

Probable Questions

Module 1

1. Differentiate between comp architecture and organisation
2. Functions of a computer with diagram
3. Von Neumann architecture -> with diagram -> structure of IAS 5m small diagram, 10m both diagrams and explanation
4. What is a bus, types of bus (dedicated and multiplex), functions of bus
5. Draw and explain bus interconnection scheme
6. What is the single bus problem and how to overcome it
7. What is bus arbitration
8. Write short note on PCI and SCSI bus with diagram

Module 2; theory is 2M

1. Booth's recoding and booth's algo difference
2. How is booth's recoding better than booth's algo?
3. What is the difference between restoring and non restoring division-2M
4. Draw flowchart for restoring or non restoring and hence solve… (3marks for flowchart) (5ish marks for solving)
5. Write the algo for division of signed numbers and hence solve so and so
6. Draw formats for IEEE single and double precision, hence solve so and so
7. Floating point + - / *, draw flowchart and solve

Module 3

1. CPU architecture draw and explain in detail
2. Draw and explain registers of 8086 process
3. Flag register: draw and explain
4. Flag register contents are given, decode them
5. Explain addressing modes with example
6. An example is given, identify its addressing mode and explain it
7. Explain the working of an instruction cycle.
8. Explain all micro-operation cycles. (Fetch, interrupt, indirect, execute)
9. Explain the functioning of micro-programmed control unit.
10. Draw the flowchart and explain the implementation of microprogrammed control unit.
11. State and explain the applications of micro programming.
12. Differentiate between risc and cisc (w program fragment)
13. Draw and explain RISC pipelining with all stages.

Module 4

1. What is memory hierarchy?
2. Characteristics of memory: LCUAPTO
3. What is dynamic RAM? Draw and explain structure. Write its features.
4. Draw and explain Static RAM structure.
5. Explain cache memory. What are its types? Explain their working in short.
6. Consider a cache consisting of 256 blocks of 16 words each for a total of 4096(4KB) words and assume that the main memory is addressable by a 16 bit address and it consists of 4KB blocks of 16 words. How many bits are there in each of the TAG,BLOCK/SET and WORD field for Direct Mapping , Fully Associative and 2-way set associative techniques?
7. A block set associative cache memory consists of 128 blocks divided into four block sets. The main memory consists of 16,384 blocks and each block contains 256 words. i) How many bits are required for addressing the main memory?(22) ii) How many bits are needed to represent the TAG, SET and WORD fields?(9,5,8)
8. A block set associative cache memory consists of 64 blocks divided into four block sets. The main memory consists of 4096 blocks and each block contains 128 words. i) How many bits are there in main memory? (19)ii) How many bits are needed to represent the TAG, SET and WORD fields?(8,4,7)
9. Perform all replacement (FIFO, LRU, OPT) algorithms
   7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
   1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6
   https://www.youtube.com/watch?v=8rcUs5RutX0

   Hit ratio : $\frac{number\ of\ hits}{number\ of\ hits + number\ of\ misses}$
10. Explain cache coherence. What are the hardware and software solutions to cache coherence? Explain snoopy protocol.
11. Explain the write policies and hence MESI protocol.
12. What is virtual memory, why is it needed?
13. Explain paging, what is a better alternative to paging?
14. RAID: all 6 levels

Module 5:

1. Functions of I/O module + diagram.
2. What are programmed and interrupt driven I/O? Which I/O is used to overcome their limitations? Explain in detail. Draw flowcharts.
3. Explain DMA transfer modes. List its advantages and disadvantages.

Module 6:

1. Explain Flynn's classification in detail [all 4]

2. What is pipelining and how does it affect CPU performance? Discuss its stages.
3. Explain pipeline hazards, their types, and methods to handle them. Also describe major design issues and principles of designing a pipelined processor.
4. Instruction sets ka program code

Via sheetal maam:

Module-6
1) Approaches for multiprocessor systems
2) need for multiprocessor
3) what is flynns classification and how multiprocessor are categorized (with diagram and eg)
4) what is pipelining/need of pipelining
5) explain 6 stage instruction pipelining(draw diag and algo too)
6) advantages of pipelining
7) what are the pipeline hazards and how to overcome it
8) what are design issues of pipeline arch?
9) what are the principle of pipeline designing processers
10) write any 5 instructions of 8086(if specified is given then explain it with eg)
11) write 8086 program.

Module 5
1) what are the problems in input output
2) what is the need of input/output module and what functions are performed by them
3) draw the block diagram of input/output module and explain each
4) what are the input/output techniques/ each is given and explain it
5) differentiate between programmed,interrupt driven and dma
6) what is the need of dma,explain with diagram and discuss dma data transfer modes.
7) advantages and disadvantages of dma.

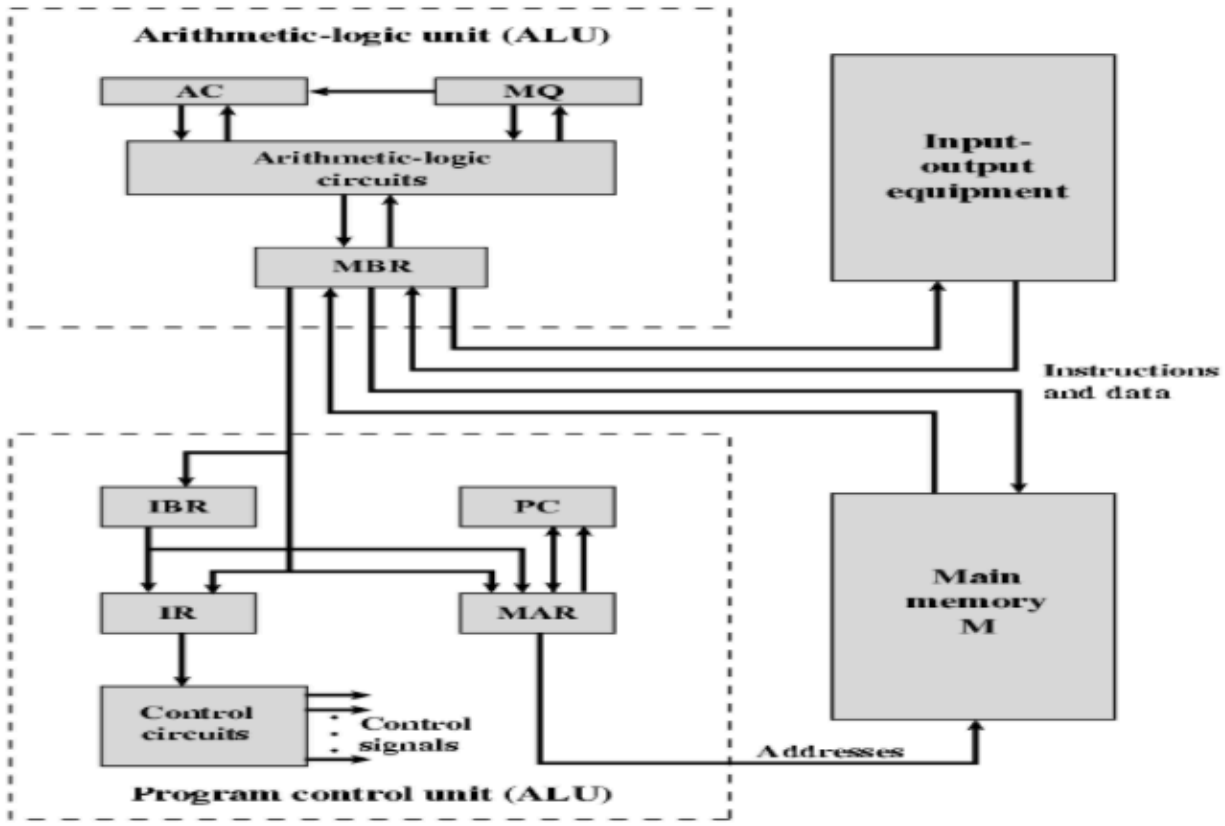
Module 4
1) what are the characteristics of memory system?
2) what is memory hierarchy and explain it in detail(5mks)
3) what is RAM,explain the structure of static ram and dynamic ram(with diag)
4) diff between static and dynamic ram
5) what is rom,where it is used(ans-system programs,os etc),explain types of rom

6) what is the need of cache memory
7) what is locality of reference/cache memory works on which principle
8) what are cache design principles
9) replacement algo me sequence will be given find-fifo lru etc
10) which of the replacement algo is best and why?
11) what are the cache mapping techniques
12) problems on cache mapping
13) what are the write policies
14) what is cache coherence problem and what are the solutions/how to overcome
15) explain mesi protocol in detail
16) short note on high speed memory(both)
17) what is virtual memory
18) how logical add get converted to physical add and explain (with diag)
19) what is paging
20) what is segmentation
21) what is TLB
TLB = Translation Lookaside Buffer.
A small, fast cache that stores recent virtual-to-physical address translations.
It speeds up address translation in paged memory systems.
22) what are the problems faced in secondary memory
23) what is RAID, explain its levels

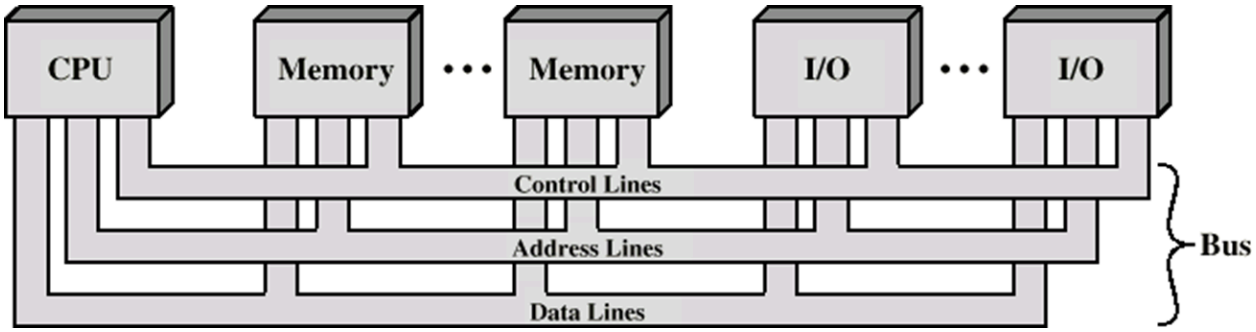
All diagrams

Von neumann

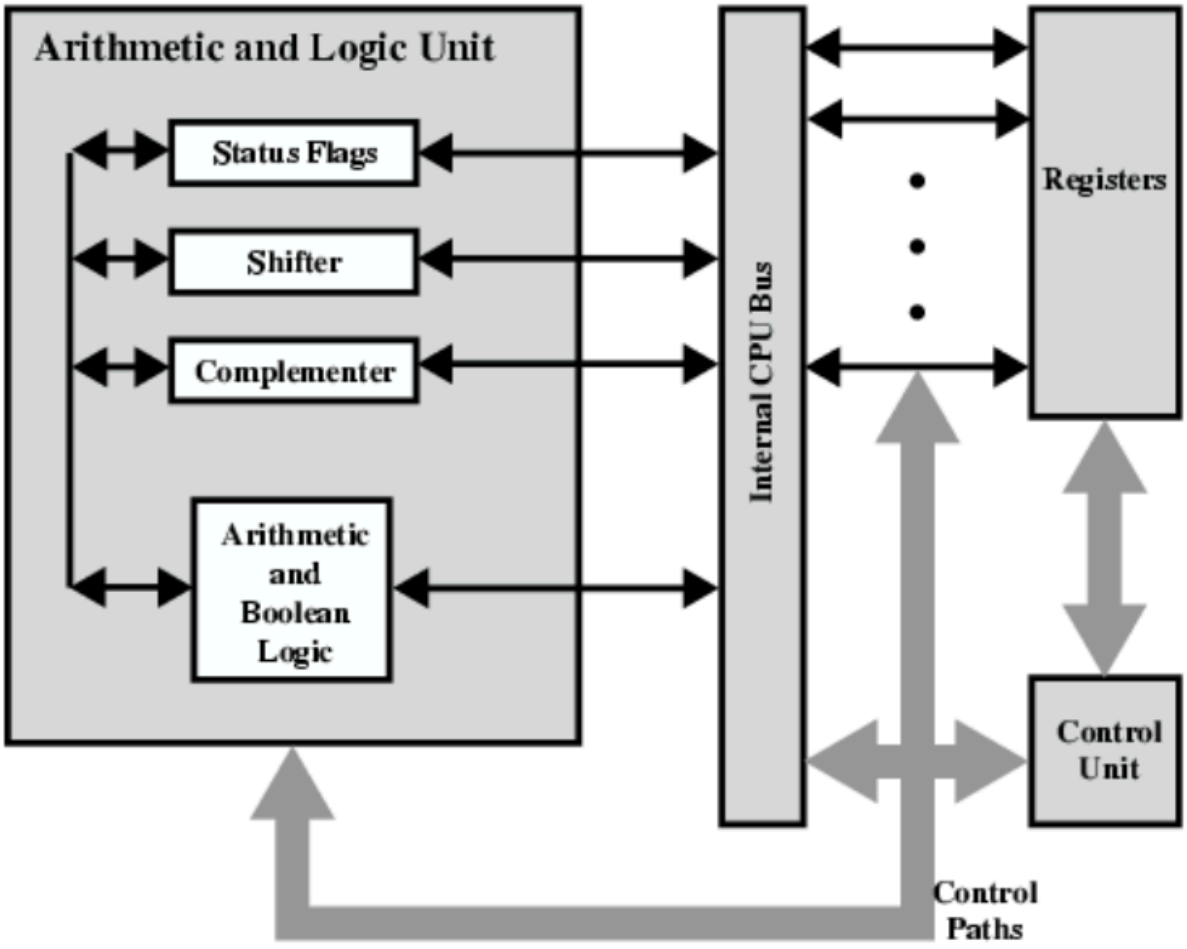
Also passes for cpu arch



Bus interconnection scheme

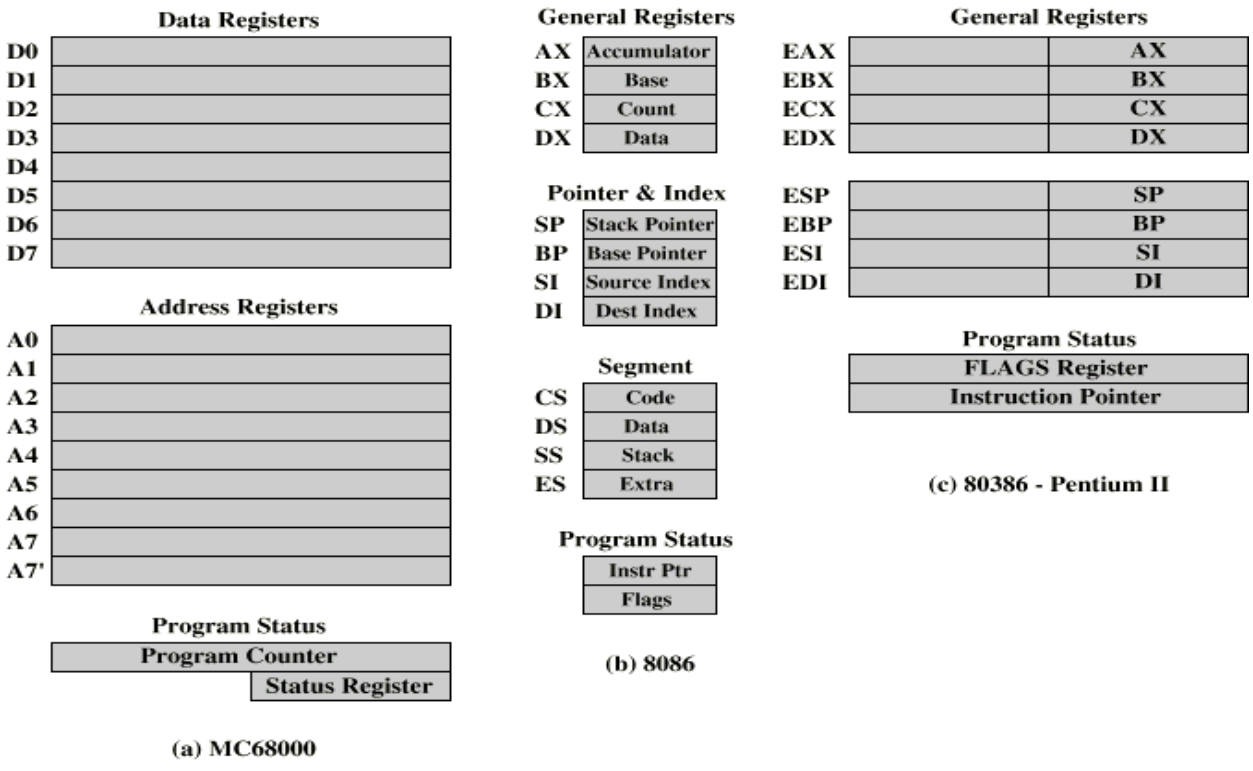


Internal CPU architecture



Registers

Only draw for whichever asked.



Flags

R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag

SF = sign flag
ZF = zero flag
AF = auxiliary carry flag
PF = parity flag
CF = carry flag

Oily dishes in Telangana stay zeroed under arms of penguins and cats.

Can give the binary representation of the flag and ask which flags are set
For 0000 0000 0000 0101 for example, cf and pf are set

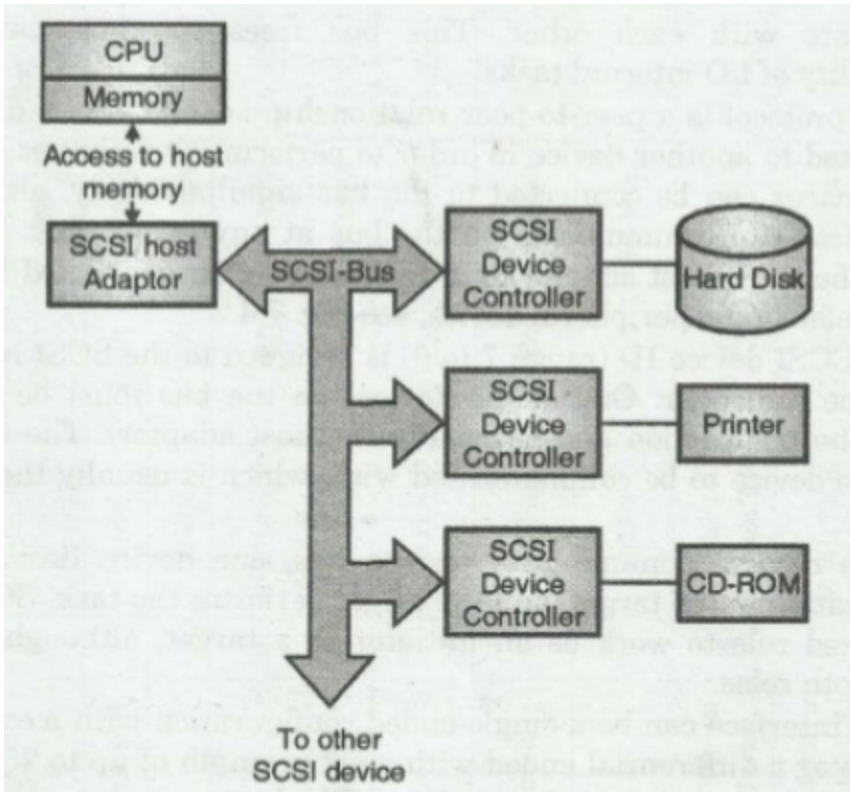
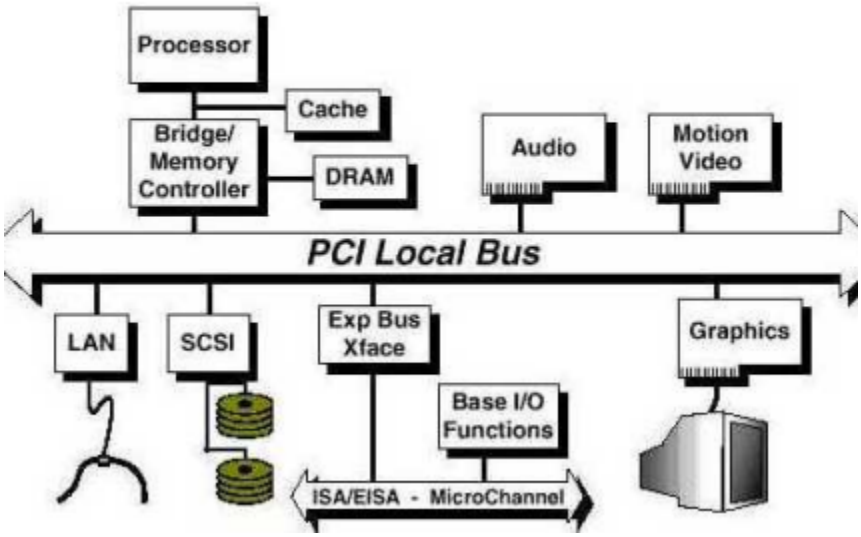


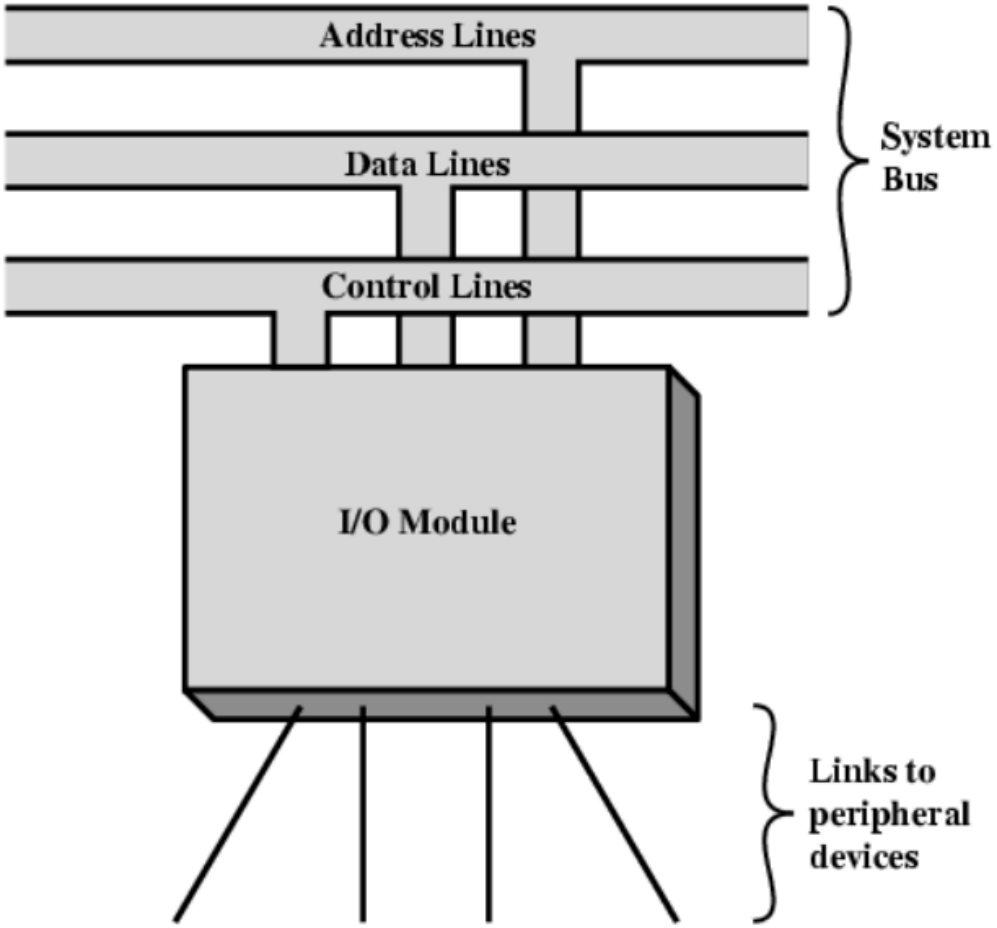
Fig. 7.4.2 : SCSI bus with host adapter and device controller scsi

5



Pci

Generic Model of I/O Module



Typical DMA Module Diagram