PES UNIVERSITY, Bangalore

Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering

UE24CS251A: DIGITAL DESIGN AND COMPUTER ORGANIZATION

Unit 3

1. **Explain the basic functions of a computer system.**

Answer:
A computer is an **electronic calculating machine** that performs the following functions:

1. **Input** – Accepts information to be processed (via keyboard, mouse, scanner, etc.).
2. **Storage** – Stores instructions and data for processing in memory.
3. **Processing** – Executes arithmetic and logical operations on the data using the ALU.
4. **Output** – Produces processed results in human-readable or machine-usable form (monitor, printer, etc.).
   These steps are coordinated by the **control unit** which ensures that the operations occur in the correct sequence

2. **Explain the role of the Memory Unit in a computer system. Differentiate between Primary and Secondary Memory.**

Answer:
The **Memory Unit** stores instructions, data operands, and results of computations.

- **Primary Memory (Main Memory):**
  - Directly accessible by the CPU.
  - Built using DRAM.
  - Volatile → loses data when power is off.
  - Examples: RAM, Cache.
  - **Advantage:** Fast access.
  - **Disadvantage:** Limited size, expensive.
- **Secondary Memory (Auxiliary Storage):**
  - Not directly accessed by CPU (data must move to primary memory first).
  - Non-volatile → retains data even without power.
  - Examples: Hard disk, CD-ROM, Magnetic tapes.
  - **Advantage:** Large storage capacity, cheaper per bit.
  - **Disadvantage:** Slower than primary memory.

3. **Explain the role of the Control Unit in the execution of a program.**

Answer:
The Control Unit (CU) is the **nerve center** of a computer that coordinates and directs the flow of data. Its functions include:

- **Instruction decoding:** Interprets the meaning of instructions fetched from memory.
- **Signal generation:** Issues timing and control signals (e.g., MEMR, MEMW, IOR, IOW) to synchronize activities.
- **Data transfer control:** Governs movement of data between CPU, memory, and I/O devices.
- **Program execution cycle:** Fetches instructions → decodes them → executes in ALU → sends results to output unit.

Without the CU, the CPU would not know **what to do next** or **when to do it**

4. **A research lab needs to perform complex weather simulations that involve trillions of floating-point calculations.**

Question:
Which type of computer should be used? Justify your answer with respect to **processing speed** and **memory hierarchy**.

Answer:

- The lab should use a **supercomputer**, since weather simulation requires **problem scalability** and huge computational speed.
- Processing is measured in **FLOPS (Floating Point Operations Per Second)** — only supercomputers provide the required performance.
- **Memory hierarchy** plays a key role:
    - **Cache + RAM (Primary memory):** Store frequently accessed simulation data for quick access.
    - **Secondary storage:** Stores massive weather datasets and historical records.
    - Without this hierarchy, even the fastest processors would stall waiting for data.
- Example: **PARAM series** in India used for climate modeling.

5. **Explain the basic operational steps of instruction execution in a computer system.**

Answer:
The CPU follows the **fetch–decode–execute cycle** for every instruction:

1. **Fetch** – The instruction is fetched from main memory using the Program Counter (PC). The PC value is transferred to the Memory Address Register (MAR), memory is read, and the instruction is loaded into the Memory Data Register (MDR), then transferred to the Instruction Register (IR).
2. **Decode** – The Control Unit decodes the instruction to determine the operation (opcode) and operands.
3. **Execute** – The ALU or other units perform the required operation. Operands are fetched from registers or memory, processed, and results are stored back in registers or memory.

4. **Increment PC** – The Program Counter is updated to point to the next instruction.
   This cycle continues until the program terminates

6. What are the functions of registers inside a CPU? Describe different types of registers.

Answer:
Registers are **fast, small storage units** inside the CPU used to temporarily hold instructions, addresses, and data.

- **Program Counter (PC):** Holds the address of the next instruction to be executed.
- **Instruction Register (IR):** Holds the instruction currently being decoded/executed.
- **Memory Address Register (MAR):** Contains the address of the memory location being accessed.
- **Memory Data Register (MDR):** Temporarily stores the data read from or written to memory.
- **General-Purpose Registers (R0, R1, …):** Store operands and intermediate results for quick ALU operations.

Registers are essential to reduce the number of slow memory accesses

7. **Differentiate between single-bus and two-bus architectures.**

Answer:

- **Single-Bus Structure:**
  - All units (CPU, memory, I/O) are connected via one common bus.
  - Only one transfer (instruction fetch or data transfer) occurs at a time.
  - Low cost and flexible for connecting devices.
  - Drawback: Bus contention, as slow and fast devices share the same line.
- **Two-Bus Structure:**
  - Uses separate buses for instruction fetch and data transfer.
  - Allows **parallelism** in fetching instructions and data.
  - Requires buffer registers to handle speed mismatch between CPU, memory, and I/O.
  - Higher cost but better performance.

8. **What is a bus in computer organization? Why are buffer registers needed in bus structures?**

Answer:
A **bus** is a collection of parallel wires that carry **data, address, and control signals** between CPU, memory, and I/O devices.

- **Data Lines:** Carry actual data.
- **Address Lines:** Specify memory/I/O locations.

- **Control Lines:** Carry read/write and timing signals.

Need for Buffer Registers:

- Devices connected to the bus operate at different speeds (CPU is fastest, peripherals are slowest).
- Buffer registers (temporary storage) smooth out timing differences by holding data during transfers.
- They prevent data loss and enable efficient transfer between fast and slow devices

9. A hospital uses a **real-time patient monitoring system**. The system continuously measures patient vitals (heart rate, blood pressure, oxygen levels) and stores them in memory. However, if a patient's oxygen level falls below a threshold, the system must **immediately alert doctors** through an alarm, even if the CPU is currently processing other tasks.

Question:
Explain how the **interrupt mechanism** helps in this situation. Which registers are affected during this process?

Answer:

- The sensor detecting oxygen drop sends an **interrupt signal** to the CPU.
- The CPU:
    1. Suspends its current program.
    2. Saves the state of execution (PC, general-purpose registers, flags).
    3. Jumps to the **Interrupt Service Routine (ISR)**, which triggers the alarm system.
    4. Once handled, restores the saved state and resumes the suspended program.
- Registers involved:
    - **Program Counter (PC):** Saved to allow resuming later.
    - **Memory Address Register (MAR), Memory Data Register (MDR):** May be used during ISR to fetch alarm instructions.
    - **General-purpose registers:** Store temporary operand values, also saved and restored.
- **Conclusion:** Interrupts ensure *immediate response* to life-critical events without constant polling by the CPU

10. **Scenario:**
An online banking server handles thousands of transactions per second. Each request involves **reading account details from disk, performing arithmetic (balance updates), and writing results back**. The system uses a **single-bus structure**, but performance issues arise as the number of requests increases.

Question:
Why is the **single-bus structure** becoming a bottleneck in this scenario? Suggest how a **two-bus architecture** could improve performance.

Answer:

- **Single-bus limitations:**
    - Only one transfer (either instruction fetch or data transfer) occurs at a time.
    - With multiple requests, CPU, memory, and I/O devices must wait for bus access, creating a bottleneck.
    - Disk operations (slow) delay faster CPU-memory interactions.
- **Two-bus advantage:**
    - One bus is dedicated to **instruction fetch**, another to **data transfers**.
    - Allows overlapping: while one instruction is being fetched, data from previous operations can be transferred simultaneously.
    - Buffer registers smooth out timing differences between fast CPU and slow I/O devices.
- **Conclusion:** A two-bus architecture reduces waiting time and increases throughput, making the banking system more efficient and responsive

11. **justify**

why computers do not use sign-magnitude or one's complement for arithmetic even though they are conceptually simpler.

Answer:

- **Sign-and-Magnitude Representation**:
    - Two representations of zero (+0 = 0000, –0 = 1000 in 4-bit).
    - Arithmetic operations are complicated because sign and magnitude must be handled separately.
    - Example: +5+(−5) → may give either +0 or –0.
- **One's Complement Representation**:
    - Negative numbers are formed by bitwise complement of positive numbers.
    - Still two representations of zero (e.g., 0000 = +0, 1111 = –0).
    - Addition requires "end-around carry," which complicates hardware.
- **Two's Complement Representation**:
    - Only **one zero**.
    - Same binary addition circuits work for both positive and negative numbers.
    - Subtraction can be performed as addition of a negative.

- Overflow detection is simpler.

12. **Prove that the two's complement of a number is equivalent to subtracting it from 2n2^n2n. Illustrate with examples.**

Answer:
For an n-bit system:

- Two's complement of NNN = (bitwise complement of N) + 1
- Bitwise complement of N = (2n−1)−N(2^n - 1) - N(2n−1)−N
- Therefore:

Two's complement of N=[(2n−1)−N]+1=2n−N

Example (n = 4):

- Take N=5→ binary (4-bit) = 0101
- 1's complement of 0101 = 1010
- Add 1 → 1011
- Decimal equivalent = 11
- Formula check: 24−5=16−5=11

13. **. A computer uses 8-bit words. Show all the steps to add the decimal numbers 93 and –46 using two's complement representation. Indicate whether overflow occurs.**

Answer:

Convert 93 to binary (8-bit):
93=010111012

Convert –46:

46=001011102

1's complement: 11010001

Add 1 → 11010010 → (–46)

Perform addition:

```
  01011101   (+93)
+ 11010010   (–46)
 ----------
  00101111
```

4. Result = 00101111 = 47 in decimal.
5. **Check for overflow:**
   - Operands have different signs (+ and –).

- Overflow cannot occur.

Final Answer: 93+(–46)=47, no overflow.

14. **Discuss how overflow can be detected by comparing the signs of operands and the result in two's complement addition.**

Answer:

- Rule: **Overflow occurs if two numbers with the same sign are added, but the result has the opposite sign.**
- Let X, Y be operands, S = result.
- Cases:
    1. X = positive, Y = positive, but S = negative → overflow.
    2. X = negative, Y = negative, but S = positive → overflow.
    3. If X and Y have opposite signs → overflow cannot occur.

15. **Compare binary number representation with character codes. Why is binary representation natural for computers, whereas ASCII is needed for human interaction?**

Answer:

- **Binary Representation**:
    - Computers are built on logic circuits operating on 2 voltage levels (0 and 1).
    - Natural way to represent numbers in hardware → binary system.
    - Supports arithmetic directly.
- **Character Codes (e.g., ASCII)**:
    - Humans interact with computers using letters, digits, and symbols.
    - These must be converted into binary patterns (codes) for storage and processing.
    - ASCII uses 7 or 8 bits to represent characters ('A' = 65 decimal = 01000001).

Binary is **machine-oriented**, while ASCII/Unicode is **human-oriented**, bridging the gap between low-level hardware representation and meaningful human-readable text.

16. **Explain Big-Endian and Little-Endian assignments with neat diagrams.**

Answer:

- **Big-Endian:**
    - The **most significant byte (MSB)** is stored at the lowest memory address.
    - Example: 32-bit word = 12 34 56 78h stored at address 1000 →
        - 1000: 12

        - 1001: 34
        - 1002: 56
        - 1003: 78
- **Little-Endian:**
    - The **least significant byte (LSB)** is stored at the lowest memory address.
    - Same example at address 1000 →
        - 1000: 78
        - 1001: 56
        - 1002: 34
        - 1003: 12

Application:

- Big-Endian used in Motorola 68000, PowerPC.
- Little-Endian used in Intel x86 processors.

17. **Explain how numbers, characters, and instructions are stored in memory.**

Answer:

- **Numbers:**
    - Stored as binary words.
    - Example: A 16-bit signed integer uses **bit 15** as sign bit; remaining 15 bits represent magnitude.
    - Binary 0000 0010 0001 1101 = decimal +541.
- **Characters:**
    - Stored as 1-byte ASCII codes (7-bit standard).
    - Example: Character A = 0100 0001 (65 in decimal).
    - A string like RAMYA stored as consecutive bytes in memory.
- **Instructions:**
    - Stored in memory words as binary patterns.
    - Typically divided into **opcode** (operation) + **operand field**.
    - Example: 16-bit instruction = 6-bit opcode + 10-bit operand address.

Thus, memory stores **data (numbers, characters)** and **program instructions** uniformly as binary patterns.

18. A computer system uses a **32-bit word length** with a **byte-addressable memory**. The system supports both **Big-Endian** and **Little-Endian** formats.

The system has **20 address lines**. Calculate the total size of the memory in bytes and words.

If an integer `0x12345678` is stored at starting address `1000`, show how the bytes will be stored in **Big-Endian** and **Little-Endian** format.

Demonstrate how **word alignment** affects storage by showing the addresses of the first four words.

If a character string `"GATE"` is stored beginning at address `2000`, show how it is represented in memory.

Briefly explain why byte-addressable memory and endianness conventions are important for **system compatibility**.

Ans:

1. **Memory Size**
    - Number of addresses = 2^{20}=1,048,5762 = **1 MB**.
    - Word length = 4 bytes (32 bits).
    - Number of words = $2^{20}/4=2^{18}$=262,144 words.
2. **Endian Representation of 0x12345678**
    - **Big-Endian:**

Address 1000 → 12

Address 1001 → 34

Address 1002 → 56

Address 1003 → 78

Little Endian:

Address 1000 → 78

Address 1001 → 56

Address 1002 → 34

Address 1003 → 12

3. **Word Alignment (32-bit = 4 bytes per word)**

- Word 0: Address 0–3
- Word 1: Address 4–7
- Word 2: Address 8–11
- Word 3: Address 12–15
- First four words align at addresses: **0, 4, 8, 12**.

Character String "GATE"

- ASCII codes: G = 71 (0x47), A = 65 (0x41), T = 84 (0x54), E = 69 (0x45).
- Stored in **byte-addressable memory** starting at 2000:

Address 2000 → 47 (G)

Address 2001 → 41 (A)

Address 2002 → 54 (T)

Address 2003 → 45 (E)

5. **Importance of Byte Addressability & Endianness**
    - Byte-addressable memory allows flexible access to characters and small data types without wasting space.
    - Endianness determines how multi-byte data is interpreted across systems.
    - For example, data written by a **Little-Endian (Intel)** system may be misinterpreted on a **Big-Endian (Motorola)** system unless conventions are standardized.

19. A program needs to compute

S=P+Q

where **P, Q, and S** are memory locations. Write the sequence of assembly-level instructions using **(i) three-address**, **(ii) two-address**, and **(iii) one-address** instruction formats.

Answer:

- **Three-address format:**
- ADD P, Q, S ; S ← [P] + [Q]
- **Two-address format:**
- MOVE Q, S ; S ← [Q]
- ADD P, S ; S ← [S] + [P]
- **One-address format (Accumulator-based):**
- LOAD P ; AC ← [P]
- ADD Q ; AC ← AC + [Q]
- STORE S ; S ← AC

Explanation: Instruction formats differ by how many explicit operands are encoded and whether an implicit accumulator is assumed

20. Explain the role of **condition codes** by writing the steps to execute the instruction:

ADD LOC, R0

Assume the instruction is stored at memory location **INSTR** and PC initially points to **INSTR**.

Ans:

Answer:
Steps:

1. PC → MAR (Place instruction address in Memory Address Register).
2. Issue **Read** → Memory → MDR.
3. Transfer MDR → IR and decode.
4. Extract operand address LOC → MAR.
5. Issue **Read** → operand at LOC goes to MDR.
6. Transfer operand from MDR → ALU.
7. Transfer contents of R0 → ALU.
8. Perform addition in ALU → result → R0.
9. Update **PC = PC + 1`.
10. Set flags: **Z, N, V, C** depending on result.

21. A stack-based processor supports **zero-address instructions**. Write the sequence of stack operations to compute:

$$Z=(A+B)\times(C-D)$$

Ans:

```
PUSH A

PUSH B

ADD       ; stack top = A + B

PUSH C

PUSH D

SUB       ; stack top = C - D

MUL       ; stack top = (A+B) * (C-D)

POP Z     ; store result in Z
```

22. Explain Indexed Addressing Mode. Write an assembly program to add marks of 5 students stored in array LIST, and store result in TOTAL.

Model Answer:

- Effective Address (EA) = Base + Index
- Suitable for arrays.
- Assembly snippet:

```
MOVE   #LIST, R1    ; Base address of array
CLEAR  R0           ; Initialize TOTAL = 0
MOVE   #5, R2       ; Counter = 5 students
LOOP: ADD    0(R1), R0  ; Add LIST[0]
    ADD    #4, R1     ; Move to next element
    DEC    R2
    BRANCH>0 LOOP
MOVE   R0, TOTAL
```

23. (a) Explain with an example how Auto-Increment and Auto-Decrement modes simplify stack/array operations.
(b) Write a program using Auto-Decrement mode to push values A, B, C into a stack.

Model Answer:

- Auto-Increment: Operand accessed, then register incremented automatically.
- Auto-Decrement: Register decremented first, then operand accessed → used in stacks.

Assembly snippet (Auto-Decrement Push):

```
MOVE   #STACKTOP, SP   ; Initialize stack pointer
MOVE   A, –(SP)        ; Push A
MOVE   B, –(SP)        ; Push B
MOVE   C, –(SP)        ; Push C
```

24. **Swap 2 numbers (X, Y)**

```
; Inputs/Outputs: X, Y (in memory)
; Uses: R0,R1
   LOAD   X, R0
   LOAD   Y, R1
   STORE  R1, X
   STORE  R0, Y
```

25. Explain the steps involved in the **assembly and execution of an assembly language program**. In your answer, clearly describe the roles of the assembler, loader, and directives such as ORIGIN, RESERVE, RETURN, and END. Give a short example program and trace how it is assembled and loaded into memory.

When a program is written in **assembly language**, it cannot be executed directly by the CPU. It must pass through several steps:

1. **Source Program Creation** – Programmer writes the assembly code using mnemonics, labels, and assembler directives.
2. **Assembly (Translation)** – The assembler translates mnemonics into machine instructions and builds a **symbol table** for labels.
    - Handles addressing modes (immediate, direct, indirect, etc.).
    - Replaces symbolic names with actual addresses.
    - Incorporates directives (e.g., ORIGIN, RESERVE).
3. **Object Program Generation** – Assembler produces the **object code** and adds a **header** with information such as program length, start address, and data allocation.
4. **Loading** – A **loader program** transfers the object code into memory. It uses header information to decide where to place instructions and data.
5. **Execution** – Loader branches to the start address (given by END directive) and CPU begins execution instruction by instruction.

2. Role of Assembler Directives

- **ORIGIN** – Specifies the starting address for placing instructions or data.
  Example: ORIGIN 100 → next instruction stored at memory location 100.
- **RESERVE** – Reserves a block of memory for data.
  Example: NUM1 RESERVE 400 → reserves 400 bytes starting at NUM1's address.
- **RETURN** – Tells assembler to insert a machine instruction that returns control to the operating system.
- **END** – Marks the end of the source program and specifies the start address for execution.
  Example: END START → assembler places START's address in header for loader to begin execution.

3. Short Example Program

```
SUM    EQU      200        ; SUM stored at address 200
       ORIGIN   204        ; data begins from address 204
N      DATAWORD 5          ; value of N = 5 at address 204
NUM1   RESERVE  20         ; reserve 20 bytes for array
```

```
        ORIGIN   100        ; program instructions begin from 100
START   MOVE     N, R1      ; load N into R1
        MOVE     #NUM1, R2  ; load address of NUM1 into R2
        CLR      R0         ; clear accumulator
LOOP    ADD      (R2), R0   ; add array element to R0
        ADD      #4, R2     ; move to next element
        DEC      R1         ; decrement counter
        BGTZ     LOOP       ; branch if >0
        MOVE     R0, SUM    ; store result in SUM
        RETURN              ; return control to OS
        END      START      ; end program, execution begins at START
```

4. Memory Arrangement

- **Data Section**
  - SUM → Address 200
  - N → Address 204, value = 5
  - NUM1 → Address 208 to 228 (20 bytes reserved)
- **Instruction Section**
  - START → Begins at address 100
  - Instructions placed sequentially: 100, 104, 108, … etc.

5. Execution Flow

1. Loader places **instructions** at addresses starting from 100 and **data** at addresses starting from 204.
2. Loader sets program counter to START = 100.
3. CPU begins execution:
   - Load loop counter N into R1.
   - Load base address NUM1 into R2.
   - Clear accumulator R0.
   - Iteratively add elements of NUM1 to R0 until R1 = 0.

- Store result in SUM.
- RETURN gives control back to OS.

5. 26. A university has an **electronic notice board** controlled by a microprocessor. Messages are typed using a keyboard and displayed character by character on the board. The interface uses the following registers and flags:

   DATAIN: 8-bit buffer register of the keyboard

   DATAOUT: 8-bit buffer register of the display

   SIN: Input status flag (set to 1 when a key is pressed, reset after read)

   SOUT: Output status flag (set to 1 when the display is ready for a new character, reset after write)

   Write an **assembly-language program** (using mnemonics like MOVE, ADD, BGTZ, BRANCH) to read a line of characters from the keyboard and display them on the board using **program-controlled I/O**.

   Explain how synchronization between processor and devices is ensured using SIN and SOUT flags.

   Discuss the disadvantage of this method compared to **interrupt-driven I/O**.

Answer:

- **Read** a character from keyboard when SIN = 1 (meaning DATAIN holds a valid byte).
- **Wait** until display is ready (SOUT = 1).
- **Write** that character to DATAOUT.
- **Repeat** until an end-of-line marker (say LF or CR) is read.
  This is **program-controlled I/O**: the CPU spins (busy-waits) on status flags and handles every transfer itself.

Minimal pseudo-assembly

```
; Assume: bit b3 of INSTATUS ↔ SIN, bit b3 of OUTSTATUS ↔ SOUT

; Registers: R0 = current char


READLINE:

 ; ---- Read from keyboard ----

READWAIT:

 TestBit  #3, INSTATUS      ; check SIN
```

```
Branch=0 READWAIT        ; wait until SIN = 1
MoveByte DATAIN, R0      ; R0 ← key code (SIN auto-clears)

; (Optional) If R0 == LF/CR → done
Compare  R0, #'\n'
Branch=EQ DONE

; ---- Write to display ----
WRITEWAIT:
TestBit  #3, OUTSTATUS    ; check SOUT
Branch=0 WRITEWAIT        ; wait until SOUT = 1
MoveByte R0, DATAOUT      ; send char (SOUT auto-clears)
Branch   READLINE

DONE:
RETURN
```

(Variants equivalent to the above appear in your slides with labels READWAIT/WRITEWAIT and transfers via DATAIN, DATAOUT, and flags SIN, SOUT.)

How synchronization is ensured

- **Keyboard path:** Key press → hardware places code in **DATAIN** and sets **SIN=1**. CPU polls SIN, then **reads DATAIN**, which **clears SIN**.
- **Display path:** CPU waits until **SOUT=1** (display ready), then writes to **DATAOUT**, which **clears SOUT**.
  These flags prevent overruns/underruns and align slow I/O with fast CPU.

27. system designer is evaluating two I/O schemes: **polling** and **interrupt-driven I/O**.
(a) Compare their CPU utilization when servicing a keyboard that generates 10 characters per second, assuming the CPU executes 10^6 instructions/sec.
(b) Explain why interrupt-driven I/O becomes essential in modern multiprogrammed systems.

Ans:

Polling vs Interrupt-driven I/O (utilization + rationale)

(a) CPU utilization comparison.
Keyboard: 10 chars/s → inter-arrival time = 0.1 s. CPU: 10^6 instr/s.

- **Polling:** The CPU spins for ~0.1 s per character doing status checks → $\approx 10^6 \times 0.1$ = **100,000 instructions wasted per char**, i.e., **~1,000,000 wasted instr/s** (nearly the full CPU) when characters are sparse. Hence utilization for useful work ≈ 0% in the polling intervals. This illustrates the classical *busy-wait* inefficiency.

- **Interrupts:** Assume a modest ISR cost of, say, 200 instructions per key. Then 10 keys/s × 200 = **2,000 instr/s** overhead → **0.2%** CPU. The rest is free for useful work. (Any reasonable ISR cost still yields orders-of-magnitude savings over polling.)

(b) Why interrupts are essential now.
Multiprogrammed systems rely on the CPU overlapping I/O wait with other work. Polling ties up the processor checking device flags (SIN/SOUT), while **interrupts let devices signal readiness**, so the CPU executes useful tasks and context-switches only on demand. This reduces idle cycles and scales to many devices with predictable latency.

28. Suppose two devices generate interrupts **simultaneously** in a vectored interrupt system.
(a) Explain how the system ensures only one device places its interrupt vector on the bus.
(b) Why is the **interrupt acknowledge (INTA)** signal critical in such a system?
(c) Propose what could go wrong if both devices placed their vector simultaneously.

Ans:

Simultaneous requests in vectored systems

(a) Ensuring a single bus driver.
Use a **grant/priority chain (daisy chain)** so only the highest-priority pending device **propagates INTA** and **drives the vector**; others pass the grant along only if not requesting.

(b) Role of INTA.
INTA indicates the CPU is ready to accept the vector and arbitrates the moment a single device may place it on the bus—preventing contention and aligning timing with instruction completion.

(c) What if two drive together?
You'd get **bus contention** (electrical conflict), **corrupted vector**, and undefined control-flow (jumping to a wrong ISR). Hardware interlocks (tri-state control, daisy chain) and the INTA handshake prevent this failure mode.

29. Explain the working of a DMA controller. How does it improve system performance compared to programmed I/O and interrupt-driven I/O?

Answer:

Answer:

A **Direct Memory Access (DMA)** controller allows data transfer directly between an **I/O device** and **main memory** without continuous processor involvement.

Working Steps

1. **Initialization by CPU:**

- The processor writes the **starting memory address**, **word count**, and **transfer direction** (Read/Write) into the DMA controller's registers.

2. **Bus Request:**
   - The DMA controller requests bus control by asserting the **Bus Request (BR)** line.
3. **Bus Grant:**
   - The CPU completes its current operation and responds with a **Bus Grant (BG)** signal.
4. **Data Transfer:**
   - DMA controller becomes the **bus master**, generates **memory addresses**, and transfers data directly between memory and I/O device.
   - Modes:
     - **Cycle Stealing:** DMA steals one bus cycle at a time.
     - **Block/Burst Mode:** DMA gains full control for the entire block transfer.
5. **Completion:**
   - Once the word count becomes zero, DMA deasserts BR and sets the **Done flag**.
   - If **Interrupt Enable (IE)** is set, it sends an **interrupt request (IRQ)** to inform the CPU.

Performance Advantage

Mode	CPU Involvement	Speed	Efficiency
Programmed I/O	Continuous polling	Slow	Low
Interrupt I/O	Occasional CPU handling	Moderate	Medium
DMA	Independent block transfer	Fast	High

Conclusion:
DMA greatly enhances performance by freeing the CPU during I/O operations and minimizing instruction overhead, especially in high-speed disk or network transfers.

30. What is bus arbitration? Explain centralized and distributed arbitration methods with examples.

Ans: **Bus Arbitration** is the process of determining which device will become the **Bus Master** — i.e., the controller of the data and address lines — when multiple devices (CPU, DMA controllers) request the bus simultaneously.

1 Centralized Arbitration

- Controlled by a **single arbiter** (often part of the CPU or a separate control unit).
- Devices send **Bus Request (BR)** signals to the arbiter.
- The arbiter sends a **Bus Grant (BG)** signal to one device based on a **priority scheme**.
- The selected device sets **Bus Busy (BBSY)** to indicate bus occupancy.

- Priority can be:
    - **Fixed Priority:** Nearest device has the highest priority (Daisy Chain).
    - **Rotating Priority:** Priority changes cyclically after each transfer.

Example:
In a **Daisy-Chain** connection, the **BG** signal passes sequentially through devices; the first requesting device captures it and blocks it from others.

2 Distributed Arbitration

- No central arbiter — all devices participate equally.
- Each device has a **unique ID** (e.g., 4-bit).
- When bus is requested:
    - Each device places its ID on arbitration lines (ARB0–ARB3).
    - Using **open-collector (wired-OR)** logic, the device with the **highest ID** wins the bus.
- Used in high-reliability systems like **SCSI**.

Example:
Device A (0101) and Device B (0110) both request the bus.
At the third bit, A sees a mismatch (0 vs 1) and withdraws.
B (0110) wins arbitration as it has the higher ID.

Comparison

Feature	Centralized	Distributed
Control	Single arbiter	Shared among devices
Complexity	Simple	Higher
Reliability	Single-point failure	No single point of failure
Example	Daisy chain (CPU–DMA)	SCSI bus arbitration

31. A CPU initiates a read operation on a **PCI bus** to fetch four 32-bit words from an I/O device.
During the transfer, the **target device** is temporarily not ready to supply data for the 3rd word. **Explain what happens on the bus during this condition and how PCI ensures proper data transfer.**

Ans:

 The transaction starts when the **CPU (initiator)** asserts FRAME# and places the **address** on the AD lines.

 The **target device** decodes this address and asserts DEVSEL# to acknowledge.

 During data phases, the handshake signals IRDY# (initiator ready) and TRDY# (target ready) coordinate timing.

 For words #1 and #2, both lines go low — data is transferred.

 For the 3rd word, the target keeps TRDY# high because it isn't ready.

- The CPU keeps IRDY# asserted low and waits.
- The bus enters a **wait-state**, inserting extra clock cycles until TRDY# goes low.

 Once ready, the target drives the data and completes the handshake.

 Finally, the initiator de-asserts FRAME# after the 4th word, ending the burst.

32. Assume a **processor requests data blocks** from a non-contiguous disk region using a **SCSI controller**.
Describe how the initiator and target coordinate this operation to maintain bus efficiency.

Ans: The **SCSI controller (initiator)** requests the bus and wins arbitration (highest ID).

 It selects the **disk controller (target)** using -SEL and sends a *Read* command.

 The **target** recognizes that the required disk sectors are non-contiguous and needs to perform multiple seek operations.

 It sends a *message* to the initiator indicating that it will **suspend the connection** and releases the bus (-BSY de-asserted).

 The target performs a **disk seek** and fills its local buffer with the first block.

 When ready, it **re-arbitrates**, wins the bus again, and **reselects the initiator**.

 Data from its buffer are transferred to the initiator using DMA.

 The target again suspends the connection and repeats the process for the next block.

 After all transfers are complete, the initiator receives an **interrupt** indicating operation completion.

33. A USB keyboard and webcam are connected through a hub to a host computer.
Explain how USB handles **simultaneous low-speed (keyboard)** and **isochronous (webcam)** data transfers without collisions.

Ans: The **USB uses a tiered-star topology** managed entirely by the **host controller**.

 The **root hub** divides time into 1 ms **frames** and sends a *Start-of-Frame (SOF)* packet every 1 ms.

 The host allocates specific frame slots:

- Keyboard → interrupt transfer every 8 ms (low-speed, small packets).
- Webcam → isochronous transfer every 1 ms (continuous stream).

 The **hub replicates host messages downstream**, but only the addressed device responds.

 Because USB is **host-scheduled and time-divided**, there are no direct device-to-device collisions.

 Isochronous traffic gets fixed bandwidth each frame; other transfers use remaining time.

Standard	Scenario Concept	Key Feature Demonstrated
PCI	Target wait-state during burst transfer	Asynchronous handshaking (IRDY#, TRDY#)
SCSI	Disk controller multi-sector read	Connection suspend / reselection
USB	Mixed low-speed & isochronous devices	Host-controlled frame-based scheduling

34. Explain the basic difference between Static RAM (SRAM) and Dynamic RAM (DRAM). Describe their internal structures, operation, and typical applications.

Ans:

SRAM (Static Random Access Memory):

- Each bit is stored using a **bistable flip-flop** made of 6 transistors (4 for latch + 2 for access control).
- Data remains stable **as long as power is supplied** — no need for refreshing.
- Fast access time (≈ few nanoseconds).
- **Read operation:** Word line enables transistors, allowing stored bit to drive bit lines.
- **Write operation:** New data overwrites latch through bit lines.
- **Used in:** Cache memory, CPU register files.
- **Advantage:** Very fast, stable, simple timing.
- **Disadvantage:** Large cell size → low density → high cost per bit.

- **DRAM (Dynamic Random Access Memory):**
  - Each bit stored as **charge on a capacitor**, accessed through one transistor.
  - Charge leaks → requires **refresh cycles every few milliseconds**.
  - **Read operation:** Word line enables transistor; stored charge sensed on bit line and rewritten.
  - **Write operation:** Bit line drives charge into capacitor.
  - **Used in:** Main memory of computers.
  - **Advantage:** High density, low cost per bit.
  - **Disadvantage:** Slower and needs refresh circuitry.

Summary Table:

Feature	SRAM	DRAM
Storage	Flip-flop (6T)	Capacitor + 1T
Refresh	Not required	Required
Speed	Fast	Moderate
Cost	High	Low
Use	Cache memory	Main memory

35. Explain the concept of memory organization using multiple memory chips. How can a 2M × 32-bit memory be built using 512K × 8 SRAM chips?

Ans:

- Each **512K × 8** chip stores 512K words of 8 bits each.
- To build **2M × 32**, we need:
    - **4 chips in parallel** for 32-bit word (8 × 4 = 32 bits).
    - **4 groups (columns)**, each containing **4 chips**.
- **Addressing:**
    - 2M = 2048K → needs **21 address lines** (2^{21} = 2M).
    - Lower 19 bits select a word within chip (512K = 2^{19} words).
    - Higher 2 bits select which chip group is enabled using a **2-to-4 decoder**.
- **Chip Select (CS):**
    - Enables one group of four chips at a time.
- **Data Lines:**
    - 32 data lines connected by combining 4 × 8-bit outputs.