PES UNIVERSITY, Bangalore

Established under Karnataka Act No. 16 of 2013)

Department of Computer Science & Engineering

UE24CS251A: DIGITAL DESIGN AND COMPUTER ORGANIZATION

Unit 3: Basic structure of computers, Standard I/O interface, Interrupts:

Computer Types, Functional Units: Input Unit, Memory Unit, ALU, Output Unit, Control Unit, Basic operational concepts

Digital systems form the foundation of modern computing and information processing. Unlike analog systems, which deal with continuously varying signals, digital systems operate on discrete signals, typically represented in binary form. The binary number system, consisting of the two digits 0 and 1, is ideally suited for representing the two stable states of electronic switching devices. A digital system processes information encoded in binary and performs tasks such as computation, control, data manipulation, and communication with high reliability and efficiency. The simplicity of binary representation also makes it easier to design circuits that can tolerate noise and other imperfections in physical signals. Even when voltage levels fluctuate slightly, as long as they remain within defined ranges for logical '0' and logical '1', the system functions reliably.

The importance of digital systems lies in their wide range of applications. Digital computers are used for general-purpose data processing, but specialized digital systems also exist in communication networks, process control, medical equipment, consumer electronics, and industrial automation. For example, digital systems are embedded in cars to control engine functions, in mobile phones for signal processing, and in household appliances for user-friendly features. Because of their versatility, digital systems have become essential components in virtually every domain of technology and daily life.

At the hardware level, digital systems are implemented using switching circuits. A switching circuit is built from electronic components that switch between two voltage levels, corresponding to binary 0 and 1. These circuits are capable of realizing Boolean functions, which mathematically describe how outputs relate to inputs. Basic logic gates such as AND, OR, and NOT form the building blocks of these systems. More complex gates like NAND, NOR, XOR, and XNOR extend the possibilities of implementing sophisticated operations. Circuits may be combinational, where outputs depend only on present inputs, or sequential, where outputs depend on both present inputs and the past history of inputs through storage elements like flip-flops.

he technology for implementing switching circuits has evolved significantly. Early digital circuits were built using mechanical relays and later vacuum tubes. These technologies, while functional, were bulky, slow, and unreliable. The invention of the transistor and subsequent development of integrated circuits (ICs) revolutionized digital systems, enabling the construction of small, fast, reliable, and power-efficient devices. With the growth of VLSI (Very Large-Scale Integration), entire processors containing millions of transistors can be fabricated on a single chip, making modern digital computers powerful and compact. The advantages of digital systems are numerous: they are reliable, can be programmed for a wide

variety of tasks, allow for convenient data storage and retrieval, and can be scaled to extraordinary levels of complexity.

Moving from the general idea of digital systems, the digital computer is a special-purpose digital system designed to be programmable and general in its applications. A digital computer is capable of accepting input, storing and processing data under the direction of stored instructions, and producing useful outputs. This versatility stems from its ability to store both instructions and data in memory and to execute instructions sequentially or conditionally, depending on program logic. The computer is built around a set of fundamental components that interact in a coordinated way.

The input unit provides the means to transfer information from the external environment into the computer. Information may include raw data to be processed or instructions that guide the operations of the system. Devices such as keyboards, scanners, sensors, and mice serve as typical input media. Once inside the computer, data must be retained temporarily or permanently. This function is served by the memory unit, which holds both instructions and data. Memory can be classified into primary memory, such as registers, cache, and RAM, which are fast and directly accessible by the CPU, and secondary memory, such as hard disks or SSDs, which provide long-term storage but at slower speeds.

At the heart of the computer lies the Central Processing Unit (CPU), which orchestrates all operations. The CPU consists of three primary parts: the Arithmetic Logic Unit (ALU), the Control Unit (CU), and a set of high-speed registers. The ALU is responsible for executing arithmetic operations such as addition, subtraction, multiplication, and division, as well as logical operations like comparisons and Boolean manipulations. The Control Unit interprets instructions stored in memory, generates control signals, and directs the coordinated flow of data between the ALU, memory, and input/output units. Registers provide temporary, ultra-fast storage for instructions and data currently being used by the CPU, thus speeding up overall computation.

The output unit is responsible for conveying the results of processing to the external environment. This could mean displaying results on a monitor, printing them on paper, playing sound through speakers, or sending signals to actuators in embedded systems. Output devices thus complete the cycle of communication between the computer and its users or the external environment in which it operates.

One of the most important concepts in digital computer design is the stored program concept, introduced by John von Neumann. In this model, both data and instructions reside in the same memory. The CPU operates in a cycle that repeatedly fetches an instruction from memory, decodes it to determine the operation to be performed, executes the instruction using the ALU and registers, and then moves on to the next instruction. This sequence, known as the instruction cycle or fetch–decode–execute cycle, provides the basis for all modern computer architectures. The idea that instructions themselves can be treated as data allows programs to be modified, stored, and executed flexibly, which is a cornerstone of modern computing.

The execution of a program involves multiple stages. First, the Control Unit fetches an instruction from memory and places it in the instruction register. Next, the instruction is decoded to determine the nature of the operation, the location of required operands, and the control signals necessary for execution. The ALU then carries out the operation, producing a

result that is stored either in a register or in memory. Finally, the CPU updates its control logic to fetch the next instruction. This systematic cycle continues until a halt instruction is encountered or until the program completes execution.

Basic operational unit and bus structure:

1.3 Basic Operational Concepts

A computer operates under the control of **instructions** that are stored in memory. The set of instructions available to the processor forms its **Instruction Set Architecture (ISA)**, which bridges the gap between hardware and software. Each instruction is composed of two main parts:

1. **Opcode (Operation code):** Specifies the operation to be performed (e.g., ADD, LOAD).
2. **Operand field(s):** Specify the data or addresses on which the operation acts.

For example, the instruction ADD MLOC, R0 means:

- Fetch operand from memory location MLOC.
- Add it to the contents of register R0.
- Store result back in R0.

Instruction Execution Cycle

The CPU executes instructions in a repetitive cycle known as the **fetch–decode–execute cycle**:

1. **Fetch:** The Program Counter (PC) contains the address of the next instruction. This address is placed into the **Memory Address Register (MAR)**, the memory is accessed, and the instruction is fetched into the **Memory Data Register (MDR)**, then transferred to the **Instruction Register (IR)**.
2. **Decode:** The Control Unit interprets the opcode in IR, identifying the required operation and addressing mode.
3. **Execute:** The ALU or other functional units perform the specified operation, using operands from registers or memory. Results are stored back into registers or memory as directed.
4. **PC Update:** The Program Counter is incremented to point to the next instruction unless a branch modifies it.

This cycle ensures sequential execution of a program. However, techniques like **pipelining** and **parallel execution** (introduced later in Hamacher) are used to improve performance.

Register Organization

Registers are the fastest storage elements in a computer system. Key registers include:

- **Program Counter (PC):** Tracks the address of the next instruction.

- **Instruction Register (IR):** Holds the current instruction for decoding.
- **Memory Address Register (MAR):** Stores the address of the memory location to be accessed.
- **Memory Data Register (MDR):** Temporarily holds data transferred between CPU and memory.
- **General Purpose Registers (R0…Rn–1):** Used for storing operands, intermediate results, and supporting ALU operations.

By minimizing memory access, registers significantly improve processing speed.

Interrupts

Normal execution of instructions may be interrupted by an external event. An **interrupt** is a hardware signal from an I/O device requesting immediate CPU service.

- On receiving an interrupt, the CPU saves its current state (PC, registers).
- The control is transferred to a predefined **Interrupt Service Routine (ISR)**.
- Once the service is complete, the CPU restores its state and resumes execution. Interrupts improve **efficiency** by avoiding constant polling of I/O devices.

◇ 1.4 Bus Structures

To enable communication among the **CPU, memory, and I/O devices**, the system uses a **bus structure**. A **bus** is defined as a collection of parallel lines that carry signals representing **data, addresses, and control information**.

Types of Buses

1. **Data Bus:** Transfers actual binary data between units. Width (e.g., 32-bit, 64-bit) determines how many bits can be transferred simultaneously.
2. **Address Bus:** Specifies the memory or I/O location involved in a transfer. Width determines the maximum addressable memory (e.g., 32-bit address bus → 4 GB memory space).
3. **Control Bus:** Carries control signals such as **Read, Write, Memory Enable, I/O Enable** and timing information to coordinate transfers.

Single-Bus Structure

- The simplest design where all components (CPU, memory, I/O) share a single communication bus.
- Advantage: **Low cost and easy to expand** (add new devices easily).
- Disadvantage: Only one operation can be performed at a time → leads to bottlenecks. Also, devices connected may vary in speed (CPU fast, I/O slow).

Buffer Registers and Synchronization

To handle differences in device speed, **buffer registers** are used. For example, when a slow I/O device (like a printer) interacts with a fast CPU, the buffer temporarily stores data, allowing transfers without stalling the CPU.

Multiple-Bus Structures

To improve performance, computers often adopt **two-bus or three-bus structures**:

- **Two-Bus Design:** One bus dedicated to instruction fetch, the other for data transfers. This reduces waiting time and improves parallelism.
- **Three-Bus Design (common in modern CPUs):** Separate buses for data, addresses, and control signals, or even separate buses for CPU–memory and CPU–I/O communication.

Importance of Bus Architecture

- Determines system **throughput and efficiency**.
- Impacts how quickly memory and I/O can respond to CPU requests.
- Affects **scalability**: adding faster processors or additional devices often requires improving bus structure (or replacing with interconnection networks).

◇ **Summary**

- Section **1.3 (Basic Operational Concepts):** Computers execute instructions in a fetch–decode–execute cycle, using registers (PC, IR, MAR, MDR) to manage flow and operands. Interrupts allow immediate handling of urgent I/O requests.
- Section **1.4 (Bus Structures):** Communication among CPU, memory, and I/O occurs via buses carrying data, addresses, and control signals. While single-bus designs are simple and cheap, they create bottlenecks. Advanced systems use multiple buses and buffer registers to improve performance.

Section 2.1

Number system

Introduction

Computers process information in binary form, since digital circuits operate on two stable states — represented as **0** and **1**. Each binary digit is called a **bit**, and multiple bits form words (n-bit words). Numbers, characters, and other data types must be encoded into binary to be stored, transmitted, or manipulated by the system. The most fundamental form is the **binary number system**, but there are different ways to represent **signed integers**.

2. Representation of Integers

There are **three classical methods** for representing signed integers in an n-bit word:

(a) Sign-and-Magnitude

- The **MSB (most significant bit)** acts as the **sign bit**: 0 = positive, 1 = negative.

- Remaining (n−1) bits store the magnitude.
- Example (4 bits):
  - +5 → 0101
  - –5 → 1101
- **Limitation:** Two different zeros exist: 0000 (+0) and 1000 (–0).

One's Complement

- A negative number is represented by complementing each bit of its positive counterpart.
- Example (4 bits):
  - +5 → 0101
  - –5 → 1010 (bitwise complement).
- Still has **two zeros**: 0000 (+0) and 1111 (–0).
- Addition requires **end-around carry correction**:
  - If a carry is generated beyond the MSB, add it back to the LSB.

Two's Complement

- The most widely used system in computers.
- Negative number = One's complement + 1.
- Formula: −N=2n−N-N = 2^n - N−N=2n−N.
- Example (4 bits):
  - +5 → 0101
  - –5 → 1011.
- Only **one representation of zero**.
- Arithmetic (addition/subtraction) uses the same binary adder logic as unsigned numbers.

Reason for adoption: Eliminates dual zero problem, simplifies arithmetic, and makes overflow detection straightforward.

Addition and Subtraction of Signed Numbers

Using Two's Complement

- Perform addition as if numbers are unsigned.
- Ignore any carry out of the MSB.
- Subtraction is done by adding the two's complement of the subtrahend.

Example (4-bit):

- (+7) + (–3)
    - 0111 + 1101 = 10100 → discard carry → 0100 = +4.

Overflow in Signed Arithmetic

- Overflow occurs when the result exceeds the representable range.
- Detection rule:
    - If both operands have the same sign but the result has the opposite sign → overflow.
- Example: In 4 bits, +7(0111)+(+3)(0011)= –6 → overflow.

Sign Extension

- When increasing the word size of a signed number in two's complement, the **sign bit must be replicated**.
- Example:
    - 4-bit (–3) = 1101.
    - Extend to 8-bit: 11111101.
- Preserves numerical value across wider registers.

Representation of Characters

- In addition to numbers, computers must process **textual information**.
- Characters (letters, digits, punctuation) are represented by **codes**.
- **ASCII (American Standard Code for Information Interchange):**
    - 7-bit or 8-bit codes.
    - Example: 'A' = 65 (01000001), 'a' = 97 (01100001), '0' = 48 (00110000).
- Characters are stored, transmitted, and manipulated like binary numbers but interpreted differently by software.

Summary

- **Sign-and-Magnitude:** Simple, but two zeros.
- **One's Complement:** Bitwise inversion, still two zeros, end-around carry.
- **Two's Complement:** Most efficient; unique zero, uniform arithmetic.
- **Overflow detection** relies on sign comparison.
- **Characters** are stored using standard codes like ASCII, extending binary representation to human-readable symbols.

2.2 Memory location and address

Main memory in a computer system is a collection of storage cells built from semiconductor technology. Each cell is capable of storing a single binary digit, either a 0 or a 1, and these cells are organized into larger groups called words. A word represents the basic unit of information that can be addressed in memory, and it may store data values such as numbers and characters or program instructions in binary form. The size of a word depends on the architecture of the machine and is typically in the range of 16 to 64 bits in modern systems. Since 8 bits equal 1 byte, many systems also define memory capacity in terms of bytes. For example, a computer with a 32-bit word length is capable of handling four bytes at a time, whereas a 64-bit computer can operate on eight bytes simultaneously.

Each word in memory must be uniquely identified so that the processor can retrieve or store information as needed. This identification is provided by the memory address, which is a numerical label assigned to every word. To select one out of the many words in memory, a set of electrical signal lines called address lines is used. The number of address lines is denoted as KKK, and it determines the size of the address space. With K address lines, the processor can generate 2K2^K2K unique addresses, meaning that memory can contain 2 pow K distinct words. The address space is therefore defined as the set of valid addresses ranging from 0 to 2^K-1. For example, if a memory system uses 10 address lines, it can access 2 power 10=1024 locations, which is 1K words. This exponential growth illustrates how even a modest increase in the number of address lines can greatly expand the capacity of memory.

It is important to distinguish between word-addressable and byte-addressable memory organizations. In a word-addressable system, each memory address refers to an entire word. For instance, in a 16-bit word-addressable computer, an address corresponds directly to a 16-bit value. In contrast, in a byte-addressable system, each address refers to a single byte, and a word consisting of multiple bytes occupies multiple consecutive addresses. Modern architectures typically employ byte addressing, which provides greater flexibility in accessing small data units such as characters. For example, in a computer with 32-bit words, successive words may begin at addresses 0, 4, 8, 12, and so on, since each word requires four byte locations.

Another important concept in memory systems is the convention of **endianness**, which determines how bytes of a word are ordered in memory. In **big-endian** assignment, the most significant byte (MSB) is stored at the lowest address of the word, and the remaining bytes follow in order of significance. This means that the word has the same address as its MSB. Conversely, in **little-endian** assignment, the least significant byte (LSB) is stored at the lowest address. For example, the 32-bit number 0x12345678 stored at address 1000 would appear as 12, 34, 56, 78 in consecutive locations in big-endian format, but as 78, 56, 34, 12 in little-endian format. Different processor families adopt different conventions: Intel processors follow little-endian organization, while Motorola and PowerPC processors use big-endian. Endianness is crucial in system compatibility, as transferring data between processors with different conventions requires careful handling to avoid misinterpretation.

Memory alignment is another practical consideration. A word is said to be aligned in memory if it is stored at an address that is a multiple of the word size. Proper alignment ensures efficient

access, as the processor can fetch the word in a single memory cycle. Misaligned words may span across two memory blocks, forcing the processor to perform multiple accesses, thereby slowing down performance. For example, in a 32-bit word-aligned system, words are stored beginning at addresses 0, 4, 8, 12, etc., ensuring each word begins at a multiple of four. In a 16-bit system, words start at addresses 0, 2, 4, 6, and so on.

Beyond numbers, memory must also be capable of storing characters and instructions. Characters are usually represented by standard encodings such as ASCII, where each symbol requires 7 or 8 bits. A string of characters is therefore stored as a sequence of bytes in consecutive addresses. For example, the string “DDCO” would be stored in memory at five consecutive byte addresses, one for each character. Instructions, on the other hand, are also stored as binary words and typically contain two fields: the operation code (opcode) that specifies the action to be performed, and the operand field that identifies data or addresses involved in the operation. For instance, in a 16-bit instruction format, the opcode may occupy 6 bits while the operand field takes the remaining 10 bits. The uniform treatment of data and instructions as binary patterns allows memory to serve as a shared storage unit for both program and data.

In summary, Section 2.2 emphasizes that memory organization revolves around the concepts of storage cells, words, and addresses. The capacity of memory is determined by the number of address lines, and the choice between word- and byte-addressable designs impacts how data is stored and accessed. Endianness conventions define how multi-byte words are represented in memory, while alignment rules influence access efficiency. Together with the ability to store numbers, characters, and instructions in binary form, these principles form the foundation of memory systems in modern computer architecture.

Section 2.4_

Instruction and sequencing

An **instruction** is a command given to the processor to perform a specific operation on data. A **program** is a sequence of such instructions organized to achieve a desired task. The central principle of modern computers is the *stored program concept*, first articulated by von Neumann, where instructions are stored in memory alongside data, and are fetched and executed sequentially. The execution of programs involves two essential tasks: specifying **what operation is to be performed** and **on which operands**. An operand can reside in a processor register, main memory, or even an input/output register.

Computer systems must therefore support a broad set of operations that fall into four categories: **data movement**, **data storage**, **data processing**, and **program sequencing and control**. Data movement refers to input and output transfers, as well as transfers within the CPU. Data storage focuses on preserving information in memory or registers. Data processing encompasses arithmetic and logical operations performed by the ALU. Finally, sequencing and control mechanisms such as branching and conditional execution determine the order in which instructions are carried out.

2. Register Transfer Notation (RTN)

In order to describe how instructions operate at a low level, **Register Transfer Notation (RTN)** is used. This notation represents data transfers between registers, memory locations, and I/O units using symbolic names. For example, processor registers are often denoted as R0, R1, R2, …, while memory locations may be labeled A, B, LOC, SUM, and so on.

The operation of copying data from a memory location M into register R0 is expressed as:

R0 ← [M]

Here, the left arrow (←) denotes assignment, while square brackets indicate the *contents* of a location. This corresponds to the assembly instruction:

MOVE M, R0

Similarly, the instruction:

ADD R3, R1

represents the operation:

R1 ← [R1] + [R3]

Thus, RTN provides a formal way to express what happens at the hardware level when instructions are executed.

3. Assembly Language Notation

At a higher level, assembly language provides **mnemonics** for opcodes and symbolic labels for operands. For example, an arithmetic expression such as:

S = P + Q

can be expressed in RTN as:

S ← [P] + [Q]

and in assembly as:

ADD P, Q, S

Here, the sum of the values at memory locations P and Q is computed and stored in S. Assembly notation reduces the complexity of machine code while maintaining a close correspondence with hardware execution.

Registers and memory operands may appear as explicit fields in instructions. For instance:

ADD R1, SUM

indicates that the content of register R1 is added to the content of memory location SUM, with the result stored back in SUM.

4. Basic Instruction Formats

Instructions can be categorized by the **number of address fields** explicitly specified.

1. **Three-Address Instructions**
   These instructions specify two source operands and a destination operand. For example:
2. ADD P, Q, S

corresponds to the operation S ← [P] + [Q]. This format provides flexibility but requires more bits to encode.

3. **Two-Address Instructions**
   These instructions use one operand as both a source and a destination. For example:
4. ADD P, Q

results in the operation Q ← [P] + [Q]. To compute S = P + Q, we must use an intermediate move instruction:

MOVE Q, S

ADD P, S

5. **One-Address Instructions**
   In this case, the second operand is implied to be the **Accumulator (AC)** register. For example:
6. LOAD P ; AC ← [P]
7. ADD Q ; AC ← AC + [Q]
8. STORE S ; S ← AC

This format saves instruction length but limits flexibility.

9. **Zero-Address Instructions**
   Used in **stack-based computers**, where operands are implicitly at the top of the stack. For example:
10. PUSH P
11. PUSH Q
12. ADD
13. POP S

Here, addition is carried out on the top two stack elements, and the result is pushed back. This format eliminates address fields entirely.

5. Instruction Execution and Sequencing

Execution of instructions typically proceeds in **straight-line sequencing**: the Program Counter (PC) holds the address of the next instruction, which is fetched, decoded, and executed. After execution, PC is incremented to point to the next sequential instruction.

The process involves two distinct phases:

1. **Instruction Fetch:** The instruction is fetched from memory into the Instruction Register (IR).
2. **Instruction Execution:** The control unit decodes the opcode and initiates the required micro-operations, such as data transfer, arithmetic computation, or branching.

Straight-line sequencing works for most instructions but is disrupted by **branch instructions**, which load a new address into the PC instead of simply incrementing it.

6. Branching and Control Flow

Branch instructions modify the normal flow of control.

- **Unconditional branches** always alter the sequence, e.g., JUMP LOOP.
- **Conditional branches** alter flow only if certain **condition codes** (flags) are set, e.g., BRANCH >0 LABEL.

Condition codes such as **Zero (Z)**, **Negative (N)**, **Carry (C)**, and **Overflow (V)** are updated after arithmetic or logical operations. They allow decision-making at the instruction level. For instance, if an addition results in zero, the Zero flag is set, and subsequent conditional branches can test this flag to direct execution.

Branching enables the implementation of **loops, if-else constructs, and subroutines** in assembly language.

7. Generating Memory Addresses and Addressing Modes

Programs frequently access arrays and lists stored in memory. To enable efficient iteration, addressing modes are used. A base register may hold the starting address, while indexing allows offsets to access different elements.

For example, to sum an array:

```
LOAD NUM1, R1   ; Base address
LOAD #N, R2     ; Counter
CLR R3          ; Accumulator

LOOP: ADD [R1], R3
INC R1
```

DEC R2

BRANCH >0 LOOP

Addressing modes like **immediate, direct, indirect, indexed, and relative** provide flexibility and reduce the number of instructions needed for complex memory access patterns.

Summary

Section 2.4 highlights the **hierarchy from instruction to program**, introducing the concept of register transfer notation, assembly language notation, instruction formats, sequencing, and branching. The different instruction types (three-, two-, one-, and zero-address) balance flexibility with instruction length and hardware efficiency. Sequencing ensures orderly execution, while branching and condition codes provide the control mechanisms that transform straight-line code into structured programs. Addressing modes further enrich instruction capability by enabling compact and efficient memory referencing.

Ultimately, this section illustrates the bridge between **hardware micro-operations** and **program-level constructs**, showing how machine-level instructions map onto the high-level programming abstractions that drive all modern computation.

2.5 Addressing modes

In any computer system, instructions are used to specify the operations that must be carried out by the processor. One of the most fundamental requirements in such an instruction is how the operands (data items) are to be located in memory or registers. The different techniques that describe the way an operand's address is specified are collectively known as **addressing modes**. The addressing mode essentially defines a rule for interpreting or modifying the address field of an instruction before the operand is actually accessed. Without such modes, the processor would be restricted in its ability to handle data efficiently, especially when dealing with large memory spaces, arrays, pointers, or looping constructs. Thus, addressing modes provide both flexibility and programming convenience.

The importance of addressing modes becomes clear when considering program control and looping structures. For example, in iteration, it is often necessary to access consecutive elements of an array or a sequence of numbers in memory. Addressing modes like **indexed** or **auto-increment** allow such operations without requiring extra instructions for pointer adjustments. Another motivation for providing multiple addressing modes is the **instruction format limitation**. Since instruction length is finite, the operand field often cannot directly accommodate large memory addresses. In such cases, alternative modes like **indirect addressing** solve the problem by allowing the instruction to point to another location that contains the actual operand address.

Immediate Addressing

In immediate addressing mode, the operand itself is specified directly in the instruction rather than its address. In other words, the instruction does not point to a memory location but carries the actual constant to be used during execution. For example, an instruction such as ADD #5, R0 adds the constant value 5 to the contents of register R0. This mode is simple and fast because no memory access is needed beyond the instruction fetch. However, it has the limitation that the size of the operand must fit within the instruction format itself, which restricts the range of constants that can be represented.

Direct (Absolute) Addressing

Direct addressing mode, sometimes called **absolute addressing**, explicitly specifies the memory location of the operand within the instruction. For instance, LOAD A, R1 means that the contents of memory location labeled A are moved into register R1. This approach is easy to use and requires only one memory reference in addition to fetching the instruction. However, the size of the address field in the instruction limits the range of accessible memory locations. Despite this drawback, direct addressing remains widely used in assembly-level programming because of its clarity and simplicity.

Register Addressing

In register addressing mode, the operand is located in one of the processor's registers, and the instruction specifies the register name. For example, ADD R1, R2 adds the contents of register R1 to register R2. This is one of the fastest modes since registers are inside the CPU and can be accessed without the delays associated with memory references. Furthermore, the instruction length is small, since only a few bits are required to identify the register. The major limitation of this mode is the small number of registers available in most processors, which restricts the amount of data that can be stored and manipulated directly.

Indirect Addressing

Indirect addressing mode introduces an additional level of referencing. Here, the instruction specifies a register or memory location that contains the effective address of the operand. For example, in LOAD (R1), R0, the contents of register R1 are interpreted as the memory address of the operand, and that operand is loaded into R0. This allows access to a much larger range of memory locations using smaller instruction fields. It is extremely useful in implementing **pointers** and **dynamic data structures**. The disadvantage is that at least two memory references are required—one to fetch the address and another to fetch the operand itself—making it slower than direct or register addressing.

Indexed Addressing

Indexed addressing mode is designed primarily for accessing arrays and tables. In this mode, the effective address of the operand is generated by adding a constant (called the displacement or offset) to the contents of an index register. Symbolically, it is written as X(R), meaning the effective address is equal to constant X plus the contents of register R. For example, ADD 20(R1), R2 will access the operand at memory address equal to 20 plus the contents of R1 and add it to R2. Indexed addressing provides high flexibility: the base of the array can be fixed, and different elements can be accessed by modifying the index register. Moreover, if the array is relocated in memory, only the base address needs to be updated, leaving the rest of the program unchanged. The drawback is the additional hardware required to compute the effective address at run-time.

Relative Addressing

Relative addressing mode is a variant of indexed addressing where the **program counter (PC)** serves as the base register. In this mode, the effective address is calculated by adding a displacement value to the current PC. This is especially useful for specifying target addresses in branch and jump instructions. For instance, BRANCH +20 means that the program should continue execution from the location 20 bytes ahead of the current instruction. Relative addressing simplifies program relocation because branch targets are expressed relative to the PC, making programs position-independent.

Auto-Increment and Auto-Decrement Addressing

Auto-increment and auto-decrement addressing modes are often used in stack manipulation and iterative operations. In **auto-increment mode**, the instruction specifies a register whose contents give the effective address of the operand, and after accessing the operand, the register is automatically incremented to point to the next memory location. For example, (R2)+ refers to the operand at the address contained in R2, and then R2 is updated to R2+1 (or R2+word size). Similarly, in **auto-decrement mode**, the register is first decremented and then used as the effective address. This is denoted as –(R2). These modes eliminate the need for separate increment or decrement instructions, thereby reducing code size and improving execution speed. They are particularly useful in implementing **stacks, queues, and iterative array access**.

Advantages of Addressing Modes

The provision of different addressing modes offers multiple benefits. Firstly, they expand the range of memory that can be addressed without increasing the instruction length, thereby saving space. Secondly, they provide flexibility to programmers by supporting constructs such as counters for loops, indexing into arrays, and pointer-based memory access. Thirdly, they enhance programming convenience by reducing the number of explicit instructions needed for common tasks like incrementing pointers or handling procedure calls. In effect, addressing

modes bridge the gap between hardware constraints (limited instruction size, register count) and programming needs (large memory space, flexible data structures).

Conclusion

Addressing modes are an essential aspect of instruction set architecture. They define how the operand of an instruction is interpreted, and they enable processors to access data stored in registers, memory, or provided directly within the instruction. From immediate and direct modes for simple operations, to indirect, indexed, relative, and auto-increment/decrement modes for complex data structures, addressing modes provide a rich set of mechanisms to support efficient programming. Carl Hamacher emphasizes that a good set of addressing modes enhances the power of the machine language while keeping hardware complexity manageable. For a student of computer organization, understanding addressing modes is critical since they illustrate the intimate connection between **instruction design, memory access patterns, and execution efficiency**.

2.6.Assembly language:

Programming in pure machine language is error-prone and tedious, since it requires specifying every operation in binary code. To make programming easier, symbolic notations were introduced, giving rise to assembly language. Assembly provides mnemonic names for operations and symbolic labels for memory addresses, while still maintaining a close relationship with machine instructions. Section 2.6 explains the structure of assembly programs, the role of the assembler, directives, and the steps in translating a program into machine code.

Assembly Language Basics

An assembly language consists of:

- Mnemonics: Symbolic abbreviations for opcodes (e.g., LOAD, ADD, MOVE).
- Operands: Registers, memory addresses, or constants involved in the operation.
- Labels: Symbolic names attached to memory locations or instructions.

Example:

MOVE R0, SUM

Here, MOVE is the opcode mnemonic, R0 is the source register, and SUM is a symbolic memory label.

Thus, assembly language enables programmers to use human-readable instructions instead of raw binary patterns.

Assembly Syntax and Statement Structure

Each line of assembly program is typically divided into four fields:

1. Label – Optional name for a memory location or instruction (e.g., START, LOOP).
2. Operation – The mnemonic opcode (e.g., ADD, MOVE, DEC).
3. Operand(s) – Registers or memory references required by the operation.
4. Comment – Non-executable text for documentation.

For example:

```
LOOP   ADD   (R2), R0   ; Add array element to accumulator
```

Addressing Modes in Assembly

Assembly allows several ways of specifying operands:

- Immediate Addressing: Operand value is given directly (e.g., ADD #10, R1).
- Direct (Absolute) Addressing: Operand is in a specific memory location (e.g., MOVE SUM, R0).
- Indirect Addressing: Operand address is found in a register or memory cell (e.g., ADD (A), R1).

Each mode provides flexibility in how instructions access data.

Assembler and Translation

The assembler is a translator program that converts the assembly source program into machine language (object program).

- As it scans the code, it maintains a symbol table mapping labels to actual addresses.
- Since labels may be used before being defined (forward references), many assemblers perform a two-pass translation:
    - Pass 1: Build symbol table, assign addresses.
    - Pass 2: Substitute actual addresses, generate object code.

The final object program is stored on disk with a header containing metadata such as program length, start address, and memory requirements.

Assembler Directives

In addition to instructions, an assembly program contains directives – commands to the assembler itself. These are not translated into machine code.

Key directives include:

- ORIGIN – Sets the starting memory address for instructions or data.
- DATAWORD – Allocates a word of memory initialized with a value.
- RESERVE – Reserves a block of memory (e.g., for arrays).
- EQU – Defines a symbolic constant (e.g., SUM EQU 200).
- RETURN – Inserts a return-to-OS instruction.
- END – Marks the end of the program and specifies the entry point.

Example Assembly Program

```
SUM    EQU      200
       ORIGIN   204
N      DATAWORD 5
NUM1   RESERVE  20

       ORIGIN   100
START  MOVE     N, R1
       MOVE     #NUM1, R2
       CLR      R0
LOOP   ADD      (R2), R0
       ADD      #4, R2
       DEC      R1
       BGTZ     LOOP
       MOVE     R0, SUM
       RETURN
       END      START
```

This program sums an array of numbers.

- Instructions are placed starting at address 100.
- Data begins at address 204 (N at 204, NUM1 at 208).
- Result is stored at SUM (address 200).

Memory Arrangement

- Instruction section: Stored sequentially from ORIGIN 100.
- Data section: Placed after ORIGIN 204, with reserved blocks for constants and arrays.
- Symbolic labels like START, LOOP, NUM1 are resolved into actual addresses by the assembler.

Role of Loader

After assembly, the loader copies the object program into memory using the header information. It places code and data at specified addresses, then starts execution from the entry point (START). The END directive ensures the correct initial instruction is chosen.

2.7 Input and output

Input/Output (I/O) operations are essential for any computer system since they enable interaction with the outside world through peripherals such as keyboards, displays, printers, and storage devices. The CPU, being much faster than most I/O devices, cannot directly communicate without a synchronization mechanism. Section 2.7 discusses the principles of program-controlled I/O, device interfacing, synchronization flags, and memory-mapped I/O.

Program Controlled I/O

In program-controlled I/O, the processor directly manages the transfer of data between I/O devices and memory. Since I/O devices are slower, the CPU must repeatedly check (poll) device status before initiating a transfer.

- On output (to display): The CPU sends the first character to the display, then waits until the display signals that it is ready for the next character. The CPU continues this process character by character.
- On input (from keyboard): The processor waits until a key is pressed and the corresponding code is available in a buffer register. Once signaled, the CPU reads the value into a register and resumes program execution.

This busy-waiting ensures correctness but wastes CPU cycles, since the processor is idle during the waiting period.

Device Interface and Registers

To support program-controlled transfers, devices are associated with small hardware modules called device interfaces. Each interface typically contains:

1. Buffer Registers
    - DATAIN: Holds the input character code from the keyboard.
    - DATAOUT: Holds the output character code to be displayed.

2. Control Flags
    - SIN (Set Input flag): Becomes 1 when new character data is ready in DATAIN. Automatically reset after the CPU reads the data.
    - SOUT (Set Output flag): Becomes 1 when the display is ready to accept the next character. Automatically reset after the CPU writes data to DATAOUT.

These flags act as synchronization signals, ensuring that data transfer only occurs when devices are ready.

Mechanism of I/O Transfer

The sequence of operations for input and output can be summarized as follows:

- Keyboard to Processor (Input Path):
    1. User presses a key → scan code stored in DATAIN.
    2. Flag SIN set to 1.
    3. CPU continuously checks SIN.
    4. When SIN=1, CPU reads DATAIN and flag is cleared automatically.
- Processor to Display (Output Path):
    1. CPU checks SOUT.
    2. If SOUT=1, CPU writes character to DATAOUT.
    3. Writing clears the flag SOUT to 0 until the display is ready again.

Thus, polling + flags form the core of program-controlled I/O.

I/O Driver Programs

To implement these transfers, I/O driver programs are written using assembly mnemonics. Typical loop structures include:

- Read Loop (keyboard):
- READWAIT: TestBit #3, INSTATUS
- Branch=0 READWAIT
- MoveByte DATAIN, R0

This waits until SIN=1, then moves keyboard data into R0.

- Write Loop (display):
- WRITEWAIT: TestBit #3, OUTSTATUS
- Branch=0 WRITEWAIT

- MoveByte R0, DATAOUT

This waits until SOUT=1, then writes R0 contents to display.

Memory-Mapped I/O

Instead of using special I/O instructions, some systems implement memory-mapped I/O, where certain memory addresses correspond to device registers like DATAIN and DATAOUT. This allows the same load/store instructions used for normal memory to access device registers.

For example:

MoveByte DATAIN, R0 ; Read from keyboard buffer

MoveByte R0, DATAOUT ; Write to display buffer

Status flags such as SIN and SOUT may be included in device status registers, and their bits can be tested using instructions like TestBit.

Example: Read and Write Routine

An integrated read/write routine may look like:

```
READWAIT:   TestBit #3, INSTATUS
            Branch=0 READWAIT
            MoveByte DATAIN, R0

WRITEWAIT:  TestBit #3, OUTSTATUS
            Branch=0 WRITEWAIT
            MoveByte R0, DATAOUT
```

This ensures that characters typed on the keyboard are echoed on the display.

Limitations of Program-Controlled I/O

- Inefficient CPU usage: CPU is fully occupied in polling loops instead of performing useful computations.
- Not scalable: For multiple devices, the overhead of constant polling becomes prohibitive.
- Better alternatives:
    - Interrupt-driven I/O → CPU only responds when device raises an interrupt.

- Direct Memory Access (DMA) → Transfers bulk data directly between memory and device without CPU intervention.

Section 4.1 – Accessing I/O Devices

The processor interacts with peripheral devices through a communication structure that usually involves a common bus shared with memory. This bus contains three types of lines: address lines, used to identify the memory location or I/O device; data lines, for transferring information between the CPU and the peripheral; and control lines, which specify the type of operation (read or write) and provide timing or synchronization signals. Each I/O device is assigned a unique address or port number. When the CPU wishes to communicate with a device, it places this address on the bus, enabling the correct peripheral to recognize and respond.

Two methods are employed for addressing devices: memory-mapped I/O and I/O-mapped (isolated) I/O. In memory-mapped I/O, the same address space is used for both memory and I/O devices. Thus, any instruction that can access memory, such as load and store, can also access I/O device registers. For example, a keyboard input register named DATAIN might be accessed with an instruction like MOV R1, DATAIN, which transfers the value of the input buffer into register R1. Similarly, MOV DATAOUT, R2 could place data from register R2 into the display device's buffer. This scheme simplifies programming, since the CPU does not need special instructions, but it reduces the usable memory address space.

In contrast, I/O-mapped I/O provides a separate address space for devices, requiring special instructions such as IN and OUT. For example, the Intel x86 architecture uses the instruction IN AL, 60h to read from the keyboard buffer at port 60h. This scheme preserves memory space, since I/O addresses are distinct, but it requires a slightly more complex instruction set. In hardware, the same physical address lines may be shared, with an additional control signal used to indicate whether the access is directed to memory or to I/O.

The connection of I/O devices to the CPU requires an interface circuit. This typically consists of three key elements: an address decoder, which activates the device when its unique address appears on the bus; a data register, which holds the information being transferred; and a status register, which records the readiness or error state of the device. For example, the keyboard's status register may contain a flag (SIN) that becomes 1 when a key is pressed. Similarly, the display device may have a flag (SOUT) that becomes 1 when it is ready to accept another character for display. These registers provide the mechanism through which the CPU can determine whether the device is ready for data transfer.

The simplest method of synchronization between CPU and device is program-controlled I/O, often called polling. In this approach, the CPU continuously checks the status register of the

device. For example, it repeatedly tests the SIN flag until it becomes 1, at which point the processor reads the data. Similarly, it checks SOUT before writing output data. While straightforward, this method is highly inefficient. I/O devices are typically much slower than the CPU, so the processor spends a significant fraction of its time waiting idly.

To improve efficiency, additional mechanisms were developed. Interrupts allow devices to notify the CPU when they are ready, avoiding wasteful polling. Another powerful mechanism is Direct Memory Access (DMA), which allows data transfer directly between device and memory without burdening the CPU with each byte transfer. These mechanisms reduce CPU overhead and are particularly important in modern systems with multiple high-speed devices.

In summary, Section 4.1 explains how I/O devices are accessed using different addressing schemes, how interface circuits operate, and why polling is limited. It sets the stage for more advanced synchronization methods, namely interrupts and DMA, discussed in later sections.

Section 4.2 – Interrupts

Interrupts are an efficient mechanism for handling I/O operations. Instead of relying on the CPU to continuously monitor the status of devices (as in polling), interrupts allow the device itself to alert the processor when it requires attention. This approach significantly improves CPU utilization, as the processor can execute useful instructions while waiting for I/O devices.

When an interrupt occurs, the sequence of events begins with the I/O device asserting an interrupt request (IR) signal. The CPU, upon completing the instruction currently being executed, suspends the normal program flow and saves the contents of the Program Counter (PC) and status information. Control is then transferred to a predefined memory location, which contains the starting address of the Interrupt Service Routine (ISR). The ISR is a short program that carries out the required service, such as transferring a character from a device register into memory. Once the ISR completes, the CPU executes a return-from-interrupt instruction, which restores the saved PC and resumes the interrupted program.

The delay between receiving an interrupt request and beginning the execution of the ISR is known as interrupt latency. Several factors contribute to latency, including the time required to save the current CPU state, memory access delays, and completion of the instruction in progress. Many processors include hardware support to reduce latency, such as automatically saving minimal state (PC and status registers) upon interrupt.

Handling multiple devices adds complexity. One approach is polling, where the CPU checks each device’s status register after receiving an interrupt request. This can waste time if many

devices are connected. A more efficient technique is vectored interrupts, in which the interrupting device provides a unique code, or interrupt vector, that directly identifies the ISR's address. During the interrupt acknowledge (INTA) cycle, the device places this vector on the bus, allowing the CPU to immediately branch to the correct ISR without polling.

When multiple devices request service simultaneously, a priority mechanism is required. A simple method is the daisy-chain priority scheme, where devices are connected serially, and the one closest to the CPU has the highest priority. However, this can cause starvation for devices lower in the chain. More advanced systems use programmable priority levels stored in the Processor Status Word (PSW). In such systems, higher-priority devices can preempt the servicing of lower-priority interrupts, leading to nested interrupts. This allows critical devices, such as timers, to be serviced promptly even while another ISR is executing. Care must be taken, however, to prevent excessive nesting, which can increase context-switch overhead and delay low-priority services.

Interrupts are also related to exceptions, which are synchronous events triggered by errors or special conditions during instruction execution. Unlike asynchronous I/O interrupts, exceptions occur as a direct result of program execution. Examples include illegal opcodes, divide-by-zero errors, or privilege violations. In these cases, the CPU may be unable to complete the current instruction and instead transfers control to an exception handler. Debugging tools also make use of exceptions through trace mode and breakpoints, which allow step-by-step execution and inspection of program state.

In conclusion, Section 4.2 emphasizes that interrupts solve the inefficiency of polling by enabling asynchronous device signaling. It also discusses interrupt latency, vectored interrupts, nesting, and exception handling. These concepts form the foundation of modern interrupt-driven I/O systems, which allow processors to manage multiple devices efficiently while maintaining system responsiveness and reliability.

Section 4.4 Direct Memory Access (DMA) and Bus Arbitration

Introduction to I/O Data Transfer

- The processor communicates with I/O devices using instructions such as MOVE DATAIN, R0.
- Before each transfer, the CPU must ensure that the device is **ready** either by:
    - **Polling** the device's status flag, or
    - Waiting for an **Interrupt Request (IRQ)** signal.
- Each data transfer requires several instructions (polling, incrementing addresses, updating counters), resulting in **high CPU overhead**.

- To overcome this inefficiency, **Direct Memory Access (DMA)** is used for large data transfers.

2. Direct Memory Access (DMA)

Definition

DMA is a hardware mechanism that allows data to be transferred directly between an I/O device and **main memory** without continuous processor intervention.

Key Idea

A special hardware unit called the **DMA Controller** manages the entire transfer process — it takes over the bus, handles addresses, and generates control signals.

3. Working of DMA Controller

Steps of Operation

1. **Initialization by CPU**
    - CPU provides:
        - **Starting address** of data block.
        - **Word count** (number of words/bytes to transfer).
        - **Direction of transfer** (Read/Write).
    - These are written into DMA registers.
2. **Bus Request & Grant**
    - DMA asserts **Bus Request (BR)** to the processor.
    - CPU responds with **Bus Grant (BG)** once it completes current instruction.
3. **Transfer of Data**
    - DMA becomes the **Bus Master**, generating all **control signals** and **memory addresses**.
    - Data is transferred directly between memory and I/O port.
    - Address is auto-incremented, and the word count is decremented after each transfer.
4. **Completion & Interrupt**
    - When the word count reaches zero:
        - **Done flag** is set in the status register.

- If **Interrupt Enable (IE)** bit is set, an **IRQ** is generated to inform the CPU that the transfer is complete.

4. Modes of DMA Operation

Mode	Description	CPU Role	Example Use
Cycle Stealing	DMA "steals" one bus cycle at a time from CPU	CPU slightly slowed	Disk or ADC transfers
Block/Burst Mode	DMA gets exclusive access to memory for entire block	CPU halted temporarily	Large disk or file transfers
Transparent Mode	DMA transfers only when CPU not using bus	CPU runs normally	Real-time systems

5. Registers in DMA Controller

- **Address Register:** Holds the next memory address for data transfer.
- **Word Count Register:** Keeps track of remaining words.
- **Control Register:** Contains direction bit (R/W), Interrupt Enable, and Start/Stop bits.
- **Status Register:** Indicates **Done**, **Error**, or **IRQ** status.

6. Advantages of DMA

Faster data transfer (no instruction overhead)
Frees CPU for other tasks
Reduces I/O bottlenecks
Supports high-speed peripherals like disks and networks

7. Bus Arbitration

When both CPU and DMA (or multiple DMA channels) request access to the **system bus**, a **Bus Arbitration** mechanism decides who becomes the **Bus Master**.

Key Terms

- **Bus Master:** Device currently controlling the bus.
- **Bus Request (BR):** DMA's request for bus control.
- **Bus Grant (BG):** CPU's signal giving permission.
- **Bus Busy (BBSY):** Indicates the bus is occupied by a master.

8. Centralized Bus Arbitration

Concept

A **single bus arbiter** (often part of CPU or a separate controller) decides which device gains control of the bus.

Mechanism

1. DMA sends **Bus Request (BR)** to arbiter.
2. Arbiter checks priorities and returns **Bus Grant (BG)**.
3. Device uses the bus and asserts **Bus Busy (BBSY)** during transfer.
4. When done, it releases the bus (BBSY = 0).

Priority Schemes

Type	Description	Example
Fixed Priority	Higher-priority device always wins	Daisy-chain connection
Rotating Priority	Priorities rotate after each service	Round-robin fairness

Daisy Chain Arbitration

- **BG signal** is passed serially through devices.
- The **first requesting device** in the chain captures BG and blocks it from others.
- Nearest device has **highest priority**.

9. Distributed Bus Arbitration

Concept

No central arbiter — all devices participate equally to determine who becomes the next bus master.

Working Principle

- Each device has a **unique 4-bit ID** (priority code).
- When a bus request occurs:
    - Each device places its ID on **ARB0–ARB3** lines.
    - Using **open-collector (wired-OR)** logic, the device with the **highest ID** wins.
    - Losers disable their lower-priority lines and wait.

Example

- Device A = 0101, Device B = 0110

- At third bit, A sees mismatch (0 vs 1) → withdraws.
- B wins and becomes the bus master.

Advantages

No single point of failure.
Highly reliable and scalable.
Suitable for **SCSI** and multiprocessor systems.

10. Comparison of Bus Arbitration Techniques

Feature	Centralized	Distributed
Arbiter	Single controller	All devices cooperate
Complexity	Simple	Moderate/High
Reliability	Single-point failure possible	More reliable
Example	Daisy chain (CPU–DMA)	SCSI bus system
Speed	Faster for small systems	Efficient for large multi-device systems

11. Importance of DMA and Arbitration

- **DMA** improves throughput by minimizing CPU intervention.
- **Bus Arbitration** ensures fairness and orderly access to shared resources.
- Together, they form the foundation for efficient **I/O subsystem design** in modern computer architectures.

Section 4.7 – Standard I/O Interfaces: PCI, SCSI and USB

Modern computer systems rely on standardized input/output interfaces to connect processors, memory, and peripheral devices in a uniform and scalable manner. Early systems used proprietary, processor-specific buses, which limited flexibility and interoperability. As systems became more complex, standardized buses were introduced to simplify device connection and communication. Section 4.7 of Carl Hamacher's *Computer Organization and Embedded Systems* presents three major standards that have shaped system design over the last three decades: the Peripheral Component Interconnect (PCI) bus, the Small Computer System Interface (SCSI), and the Universal Serial Bus (USB). Each of these standards represents an evolution in how computers communicate with their peripherals, balancing speed, cost, and convenience.

The PCI Bus

The PCI bus was developed in the early 1990s to provide a high-speed, processor-independent interface between the CPU, memory, and peripheral devices. Unlike earlier local buses that were tightly coupled to a particular processor architecture, PCI defines both the electrical and protocol layers, allowing it to be used with a variety of microprocessors. A PCI system typically consists of a host bridge connecting the CPU–memory subsystem to the PCI bus and several peripheral devices—such as disk controllers, network interfaces, and display adapters—connected as bus masters or slaves.

Communication on the PCI bus follows a well-defined sequence. Every transaction begins with an *address phase*, during which the initiator (usually the CPU or a DMA controller) places the memory or I/O address on multiplexed address/data lines (AD[31:0]). The target device decodes the address and signals its presence by asserting the *Device Select (DEVSEL#)* line. Subsequent *data phases* transfer one or more 32-bit or 64-bit words, synchronized by the *Initiator Ready (IRDY#)* and *Target Ready (TRDY#)* signals. Data transfer occurs only when both are asserted low, providing an asynchronous handshake that automatically introduces wait states whenever either side is not ready. Burst transfers are supported by keeping the *Frame#* line active across multiple data phases. At the end of the burst, the initiator de-asserts *Frame#*, completing the transaction.

Because several devices may need bus access, PCI includes an arbitration mechanism managed by a centralized arbiter. Devices request control through *Bus Request (BR#)* lines and receive permission via *Bus Grant (BG#)* signals. This scheme allows multiple bus masters to share the PCI bus efficiently while avoiding conflicts. The combination of multiplexed lines, asynchronous handshaking, and centralized arbitration makes PCI both high-performance and cost-effective. It dominated personal-computer architecture through the 1990s before giving way to PCI Express, which extends the same principles over a serial link.

The SCSI Bus

The Small Computer System Interface, or SCSI, was designed primarily for high-speed block devices such as hard disks, optical drives, and tape units. Whereas PCI defines a generic system bus, SCSI is a specialized I/O bus with its own controller logic and protocol. Each device on a SCSI bus is assigned a unique identifier—ranging from 0 to 7 for an 8-bit bus, or 0 to 15 for a

16-bit “wide” implementation. Communication occurs between two types of controllers: an *initiator*, which begins an operation, and a *target*, which executes it. A disk controller, for example, usually acts as the target, while a host adapter in the CPU acts as the initiator.

The SCSI protocol allows several sophisticated features that improve efficiency. Once the initiator wins control of the bus through an arbitration process (in which the highest SCSI ID takes priority), it selects a target by asserting selection signals and sending a command block that specifies the operation. After this, the target may temporarily release the bus while it performs time-consuming operations such as disk seeking or data buffering. When ready, the target re-arbitrates for the bus, reselects the initiator, and resumes data transfer. This capability to suspend and later restore a connection permits *overlapping operations*: while one disk is seeking, another device can use the bus, thereby increasing system throughput. Data are typically transferred by direct memory access (DMA), minimizing CPU involvement.

The SCSI bus uses a set of control signals such as *BSY* (busy), *SEL* (selection), *REQ* (request), and *ACK* (acknowledge) to manage handshaking and data flow. Information is transferred in distinct phases—command, data, status, and message—each indicated by control lines *C/D*, *I/O*, and *MSG*. Because of its robust handshaking, error detection, and support for multiple concurrent transactions, SCSI long served as the preferred interface in servers and workstations where high I/O bandwidth and reliability were critical.

The Universal Serial Bus (USB)

While PCI and SCSI focus on performance and parallel transfer, the Universal Serial Bus emphasizes simplicity and ease of use for everyday peripherals. Introduced in the mid-1990s, USB replaced a multitude of legacy ports—serial, parallel, PS/2, and game ports—with a single unified serial interface. It provides both data transfer and limited power delivery through the same cable, supporting *hot plugging* and *automatic configuration* so that devices can be connected or disconnected without rebooting the system.

USB uses a *tiered-star topology* in which a single *host controller* manages communication with all devices through one or more *hubs*. Each hub expands the number of downstream ports, allowing up to 127 devices on a single host. Unlike SCSI, USB communication is always initiated by the host; devices cannot communicate directly with one another. Data transfer occurs in packets that include a synchronization field, a packet identifier (PID), an address and endpoint number, data bytes, and a cyclic redundancy check.

Four distinct transfer types support different applications. **Control transfers** handle device configuration and status. **Interrupt transfers** are used for small, time-critical data such as keyboard or mouse input. **Bulk transfers** move large, non-time-critical data, as with printers or scanners, and **isochronous transfers** provide guaranteed timing for streaming audio and video. The bus is time-divided into one-millisecond *frames*, each beginning with a *Start-of-Frame (SOF)* packet generated by the host. The host schedules all transactions within these frames, allocating bandwidth according to device requirements. Isochronous transfers receive reserved slots to maintain steady data rates, while other types use the remaining bandwidth.

Electrical signaling on USB employs differential pairs (D+ and D–) and uses NRZI encoding with bit-stuffing for synchronization. Various speed grades have evolved—from the original *Low-Speed (1.5 Mbps)* and *Full-Speed (12 Mbps)* of USB 1.1 to *High-Speed (480 Mbps)* in USB 2.0 and several Gbps in USB 3.x. The protocol includes handshake packets such as *ACK*,

NAK, and *STALL* to ensure reliability and flow control. Because the host controls the entire schedule, USB can handle devices of widely differing speeds without collisions.

Comparative Perspective

Although PCI, SCSI, and USB differ in topology and purpose, they share several common design goals: standardized connectors, well-defined electrical signaling, layered protocols, and device independence. PCI integrates high-bandwidth peripherals within the system chassis, SCSI coordinates intelligent controllers for mass storage, and USB extends connectivity to external low-cost consumer devices. Collectively, these standards illustrate how modular interfaces enable scalable, interoperable, and efficient computer systems. PCI demonstrates high-speed parallel bus design, SCSI exemplifies intelligent peripheral communication with overlapping operations, and USB represents the evolution toward flexible serial, host-controlled networks. Together, they mark the transition from processor-specific wiring to universal connectivity—a hallmark of modern computer organization.

5.1 Semiconductor RAM Memories

andom Access Memory (RAM) is the working memory of a computer used for temporary storage of programs and data while the processor executes instructions. RAM is **volatile**, meaning that its contents are lost when power is turned off. Semiconductor memories are built using integrated-circuit technology and are broadly classified into **Static RAM (SRAM)** and **Dynamic RAM (DRAM)**. Both store binary information but differ in their internal structure, speed, cost, and refresh requirements.

Static RAM (SRAM) stores each bit in a *bistable latch* made from six transistors – four forming the flip-flop and two providing access control to the bit lines. Once data is written into an SRAM cell, it remains stable as long as power is supplied; no refresh circuitry is required. The read operation is performed by activating the word line to connect the selected latch to its bit lines, allowing the stored logic level to drive the output amplifiers. During a write operation, new data from the bit lines overwrites the latch state. Because the cell uses more transistors, SRAM offers high speed and reliable operation but has low packing density and is costly per bit. It is therefore used in applications where speed is critical, such as **cache memory** and **register files** within the CPU.

Dynamic RAM (DRAM), in contrast, stores information as electric charge on a small capacitor accessed through a single transistor. Each DRAM cell thus consists of only one transistor and one capacitor, resulting in very high density and low cost. However, the stored charge gradually leaks away, so each cell must be **periodically refreshed**—typically every few milliseconds. Reading a DRAM cell is destructive because the capacitor discharges when its charge is sensed; therefore, the memory controller immediately rewrites the data back into the cell after every read. Write operations involve driving the appropriate charge onto the capacitor through the access transistor. DRAMs form the **main memory** in computers, where large capacity is more important than extremely low latency.

To coordinate the operations of a DRAM chip, external control signals are used—primarily the **Row Address Strobe (RAS)** and **Column Address Strobe (CAS)**. Because a complete address cannot be applied simultaneously due to pin limitations, the address is multiplexed: the

row address is supplied first and latched on the RAS signal, followed by the column address latched on CAS. Additional control lines such as **Write Enable (WE)** and **Chip Select (CS)** determine the direction of data transfer and whether the chip is active. A 16-Mbit DRAM, for example, might be organized as 2 M × 8 bits, using a 12-bit row and a 9-bit column address within a 4 K × 4 K cell array. Internally, DRAM chips include sense amplifiers and refresh circuitry to manage data integrity automatically.

Early DRAMs were **asynchronous**, meaning their operation was not tied to the system clock; the processor or memory controller had to generate all timing signals explicitly and wait for completion before initiating the next access. Later, the development of **Synchronous DRAM (SDRAM)** aligned all operations with the CPU clock, allowing commands to be registered on clock edges and enabling **pipelined, burst-mode data transfers**. In SDRAM, once the first data word of a burst is available, subsequent words are delivered in consecutive clock cycles, significantly increasing throughput. The internal mode register defines burst length, latency, and operating parameters. SDRAM thus provides higher bandwidth than asynchronous DRAM and forms the basis of the modern **DDR (Double Data Rate)** family, where data is transferred on both rising and falling clock edges.

o further increase memory bandwidth, **Rambus DRAM (RDRAM)** was introduced. It uses a narrow, high-speed **Rambus Channel** with differential signaling and small voltage swings (±0.3 V around 2 V) to achieve effective transfer rates up to 800 MHz. Communication occurs through packets: *Request*, *Acknowledge*, and *Data* packets flow between a master (processor) and slave (RDRAM modules). RDRAM modules, called **RIMMs**, contain multiple chips connected serially along the Rambus channel. While RDRAM achieved very high data rates, its proprietary design and higher cost limited widespread adoption; open-standard **DDR SDRAM** soon replaced it due to lower price and better scalability.

Large memory systems are built by combining several smaller chips. For instance, a **2 M × 32-bit** memory can be constructed from **512 K × 8** SRAM chips. Four chips are grouped in parallel to provide the 32-bit word width, and a **2-to-4 decoder** uses high-order address bits to activate one group at a time via **Chip Select (CS)** lines. The remaining address bits select the desired word within each chip. This modular organization allows designers to scale capacity and data-bus width independently. In dynamic memory systems, similar techniques are applied, with chips mounted on **SIMM** or **DIMM** modules that can be inserted into the motherboard to expand main memory.

A **Memory Controller** serves as the interface between the processor and the DRAM array. It performs address multiplexing (splitting the address into row and column parts), generates and synchronizes RAS, CAS, R/W, and CS signals, and manages refresh cycles using an internal counter. The controller ensures that every row is refreshed within the specified time interval (for example, all 8 K rows in 64 ms). It also aligns memory timing with the CPU clock and may implement error-correction logic such as parity or **ECC (Error Correcting Code)**. With advanced DRAMs such as SDRAM, most of these control functions are integrated within the chip itself, simplifying system design.

In summary, **semiconductor RAM technologies** have evolved from simple asynchronous DRAMs to sophisticated clock-synchronized SDRAMs and high-bandwidth architectures like RDRAM and DDR. **SRAM** remains the fastest and most reliable for cache applications, while **DRAM** provides the large, economical main memory essential for modern computing. The

combination of these memory types, orchestrated by the memory controller, forms the hierarchical storage structure that balances **speed, capacity, and cost** in every computer system.