Module 4

Instruction Processing and Parallel Architectures

Learning Objectives

By the end of this module, learners will be able to:
Understand the design and components of a basic processing unit.

Explain the steps involved in instruction execution and data flow.

Describe the concept and advantages of multiple bus organization.

Differentiate between hardwired and microprogrammed control units.

Explain the fundamentals of pipelining and identify various types of hazards.

Analyze the impact of instruction hazards on instruction set architecture.

Explore parallel computer models including multiprocessors and multicomputers.

Understand the principles of PRAM and VLSI computation models.

Structure

4.1 Basic Processing Unit Design

The **Basic Processing Unit (BPU)** refers to the core components of a Central Processing Unit (CPU) that work together to fetch, decode, and execute instructions. This unit serves as the brain of a digital computer and is responsible for processing all data and controlling all operations within the system. It integrates the **data path**, **control path**, **registers**, and **arithmetic and logic components**, enabling the CPU to function efficiently. The design and organization of the processing unit directly influence the performance, complexity, and power efficiency of the processor.

4.1.1 Components of a Processing Unit

The processing unit contains several key functional blocks, each with a specific role. Together, these components carry out all operations required by instruction execution, including data manipulation, address calculation, instruction sequencing, and control signal generation.

Key Components:

Arithmetic Logic Unit (ALU):

Performs all arithmetic operations such as addition, subtraction, multiplication, and division.

Also executes logical operations like AND, OR, NOT, and XOR.

Takes inputs from registers and outputs the result to a destination register.

Often includes a status register that holds flags (e.g., carry, zero, overflow) used for conditional branching.

Register File:

A collection of **general-purpose registers** used for temporary data storage during instruction execution.

May also include **special-purpose registers** like:

Program Counter (PC) – Holds address of the next instruction.

Instruction Register (IR) – Holds the currently fetched instruction.

Accumulator (ACC) – Often used in single-accumulator machines for temporary arithmetic results.

Status or Flag Register – Holds condition flags for branching.

Control Unit (CU):

Directs operations of the processor by generating **control signals**.

Decodes the fetched instruction and issues commands to data path elements.

Coordinates the timing of operations using system clocks.

Can be **hardwired** or **microprogrammed** in design.

Internal Buses:

Serve as communication channels between components.

Data Bus: Transfers data among registers, memory, and ALU.

Address Bus: Carries memory addresses from the CPU to RAM or I/O.

Control Bus: Carries control signals and status information.

Clock Generator:

Provides clock pulses to synchronize operations.

Ensures that all components perform tasks in a timed and orderly manner.

4.1.2 Control Path and Data Path

The **control path** and **data path** are two fundamental architectural elements in any processing unit. They represent the split between components that **generate instructions (control)** and those that **perform operations (data processing)**.

Data Path:

Represents the **hardware elements that move and transform data**.

Includes:

ALU

Registers

Multiplexers

Shifters

Internal Buses

Responsible for:

Arithmetic and logical computation

Data movement between registers and memory

Operand selection through multiplexers

Control Path:

Comprises the **Control Unit** and supporting logic.

Responsible for:

Decoding instructions in the IR

Generating control signals for data path components

Coordinating register transfers and operation timing

Interaction:

Every operation involves the **control path commanding the data path**.

For example, to add contents of R1 and R2, and store result in R3:

Control unit activates signals to select R1 and R2.

Routes data to ALU and configures ALU to perform addition.

Stores result in R3 after completion.

4.1.3 Register Transfer Operations

Register Transfer Operations (RTOs) form the basis of micro-operations within the CPU. They define how data is moved between registers and how basic operations are applied.

Types of Register Transfers:

Data Transfer Operations:

Move data from one register to another.

Example: R1 ← R2 (content of R2 is copied to R1).

Arithmetic Operations:

Basic arithmetic using registers as operands.

Example: R1 ← R2 + R3 (result stored in R1).

Logical Operations:

Bitwise logical functions between registers.

Example: R1 ← R2 AND R3.

Shift Operations:

Left or right shifts for arithmetic or logical use.

Example: R1 ← R1 << 1 (logical left shift).

Micro-Operation Execution:

Micro-operations are coordinated through **clocked control signals**.

One or more RTOs may be performed during a single clock cycle depending on CPU architecture.

4.1.4 Functional Flow of the Processing Unit

The **functional flow** outlines how instructions are handled from the moment they are fetched to the point of result storage. It integrates control and data flow to define the overall behavior of the processing unit.

Stages of Flow:

Instruction Fetch:

PC points to the address of the next instruction.

Memory is accessed to fetch the instruction into the IR.

Instruction Decode:

CU interprets opcode and identifies operands and operations.

Operand Fetch:

Based on the instruction, operands are selected from registers or memory.

Execution:

ALU or other functional units carry out the operation.

Result Storage:

The result is written back to the register file or memory.

PC Update:

For sequential instructions, PC is incremented.

For branch/jump instructions, PC is modified accordingly.

4.2 Instruction Execution Steps

The execution of instructions in a processor follows a **stepwise cycle**, commonly known as the **Fetch–Decode–Execute** cycle. Each instruction undergoes several internal operations that involve accessing memory, registers, and control logic to complete its task.

4.2.1 Fetch, Decode, Execute Cycle

This cycle represents the **core operational behavior of a CPU**, where every machine instruction passes through three primary stages:

1. Fetch:

The PC provides the address of the instruction.

Instruction is fetched from memory and loaded into the IR.

PC is incremented to point to the next instruction.

2. Decode:

Instruction is analyzed to identify:

Operation (via opcode)

Source and destination operands

Addressing mode (immediate, direct, indirect)

The CU interprets the binary opcode and determines the control signals required for execution.

3. Execute:

Operation specified in the instruction is carried out.

Could involve:

ALU computation

Memory access

Control transfer (jumps, branches)

Final result is stored or used based on instruction semantics.

4.2.2 Operand Fetch and Result Storage

Instruction execution requires fetching data (operands), performing an operation, and storing the result.

Operand Fetch:

Operands may reside in:

General-purpose registers

Main memory

Instruction itself (immediate data)

The location is determined by the **addressing mode** encoded in the instruction.

Result Storage:

Once the ALU or logic unit computes the result:

It is stored in a designated register or memory location.

Destination is defined by the instruction format.

If memory write is required, the **memory address** and **data bus** are engaged.

4.2.3 Control Signal Generation

Execution of any instruction is governed by a series of **control signals** that trigger specific hardware operations.

Control Signal Types:

Read/Write: To perform memory operations

ALU Operation Codes: Define what ALU should compute

Register Enable/Disable: For operand and result movement

Multiplexer Select Lines: For choosing input sources

Load Signals: For latching data into registers

Control Unit Approaches:

Hardwired Control:

Uses fixed logic circuits (gates, flip-flops)

Fast but inflexible

Microprogrammed Control:

Stores control logic as microinstructions in control memory

Easier to modify or expand, suitable for complex instruction sets

4.2.4 Instruction Timing and Synchronization

Timing ensures that all instruction operations occur **in the correct sequence and within proper time intervals**. Synchronization is crucial in multi-step operations and pipelined processors.

Key Concepts:

Clock Signal:

Provides timing reference

Controls operation boundaries (start and stop of register transfers, ALU operations)

Timing Diagrams:

Visualize the sequence of operations with respect to clock cycles

Pipelining:

Allows multiple instructions to be in different execution stages simultaneously

Increases throughput but requires sophisticated hazard detection

Synchronization Challenges:

Handling **memory access delays**

Managing **multi-cycle instructions**

Coordinating between control unit and data path operations

4.3 Multiple Bus Organization

A bus is a **shared communication pathway** used to transfer data, addresses, and control signals between different components in a computer system. In early processor designs, a **single bus system** was used for simplicity. However, as computing demands and hardware complexity grew, **multiple bus architectures** emerged to increase **data transfer speed**, reduce **contention**, and improve **parallelism**. A multiple bus organization allows various system components to communicate simultaneously without waiting for a single shared bus, leading to better performance and modularity.

4.3.1 Single Bus vs. Multiple Bus Architecture

Single Bus Architecture:

In a single-bus system, **all components** (CPU, memory, I/O devices) are connected to one shared bus. This bus carries data, addresses, and control signals.

Characteristics:

Simple design and low cost

All transfers (CPU-memory, I/O-memory, etc.) happen **sequentially**

Only one data transfer can occur at a time

Limitations:

Bus contention: Multiple components competing for the bus cause delays

Low bandwidth: Performance bottleneck as the number of components increases

Scalability issues: Difficult to support multiple processors or high-speed devices efficiently

Multiple Bus Architecture:

Multiple bus systems introduce **additional buses** to separate different types of traffic (e.g., internal CPU buses, memory buses, I/O buses). This division increases throughput and minimizes interference.

Types of Buses in Multi-bus Systems:

Internal CPU Buses: Connect internal components like ALU, registers, control unit

System Bus: Connects CPU to memory and I/O

I/O Bus: Dedicated to input/output devices

Backplane Bus: For connecting multiple processors or boards in larger systems

Comparison Table:

Feature	Single Bus	Multiple Bus
Cost	Low	Higher
Speed	Limited	Improved
Parallel Transfers	Not possible	Possible
Contention	High	Reduced
Scalability	Poor	Better for complex systems

4.3.2 Benefits of Multiple Bus Systems

The shift toward multiple bus organizations brings several **architectural and performance benefits**, especially in high-performance and multiprocessor systems.

Major Benefits:

Increased Parallelism:

Multiple transfers can occur simultaneously on separate buses.

For example, while one bus handles memory access, another can transfer data to an I/O device.

Reduced Contention:

Data traffic is divided across buses, minimizing delays from competition for shared resources.

Improved Throughput:

More buses allow faster data exchange between units, increasing the instruction execution rate.

Better Support for Pipelining and Multicore:

Processors using **instruction-level parallelism** or **multithreading** benefit from dedicated buses for instruction and data fetching.

Scalability and Modularity:

Easier to integrate new devices or subsystems without disrupting the overall system.

Common in systems using **modular designs** like blade servers or backplane-based architectures.

Fault Isolation and Reliability:

Failures in one bus do not affect the entire system.

Maintenance and diagnostics are easier.

4.3.3 Interconnection and Data Flow Control

In multiple bus systems, **interconnection design** becomes crucial to maintain **coherent data flow** and **efficient communication** between units.

Key Concepts:

Bus Interconnection Schemes:

Hierarchical: Buses are layered (e.g., local bus connects CPU components, system bus connects CPU to memory).

Central Bus Arbiter: Coordinates access between components to shared buses.

Crossbar Switch: Allows multiple simultaneous connections between modules, reducing bus contention significantly.

Data Flow Control Mechanisms:

Ensures that data moves in correct sequence and format.

Tri-state Buffers: Prevent bus conflicts by allowing only one device to transmit at a time.

Handshaking Protocols: Synchronous or asynchronous control signals manage communication start/stop between devices.

Synchronization Challenges:

Coordinating data movement across buses running at different speeds

Managing control signal timing for safe transfers

Ensuring integrity of data transfers using parity bits or CRC

4.3.4 Bus Arbitration and Prioritization

In any shared bus architecture—even within a multi-bus system—**bus arbitration** is needed when multiple devices request access to the same bus. Arbitration ensures **orderly and conflict-free access**.

Bus Arbitration:

The process of determining **which device** gets control of the bus when **multiple requests** are made simultaneously.

Typically managed by a **bus arbiter** component, either **centralized** or **distributed**.

Types of Arbitration Techniques:

Daisy Chaining (Serial Arbitration):

Devices are connected in series.

Priority is determined by position in the chain (closer = higher priority).

Simple but unfair to lower-priority devices.

Polling:

Controller checks each device in a sequence.

Requires control logic to identify requestors.

Scalable but introduces delay.

Fixed Priority:

Each device is assigned a fixed priority.

Higher priority device always wins.

May cause **starvation** of lower-priority devices.

Rotating Priority (Round Robin):

Priority shifts among devices after each access.

Ensures fairness and prevents starvation.

Dynamic Arbitration:

Priority changes based on request frequency or time waiting.

Adaptive and fair but complex to implement.

Arbitration Signals:

Bus Request (BR): Sent by device requesting the bus.

Bus Grant (BG): Sent by arbiter to the device allowed to use the bus.

Bus Busy: Indicates whether the bus is currently in use.

4.4 Hardwired and Microprogrammed Control

The **control unit** is a critical part of a computer processor that directs the flow of data and coordinates operations within the CPU. It interprets machine instructions and generates control signals needed to execute them. There are two primary methods for implementing the control unit: **Hardwired Control** and **Microprogrammed Control**. The choice between these methods affects the **speed**, **complexity**, **flexibility**, and **modifiability** of the processor.

4.4.1 Concept of Control Unit

The **Control Unit (CU)** is responsible for **fetching instructions**, **decoding them**, and **generating control signals** that drive the operation of the processor. It acts as the

"decision-maker" of the CPU, ensuring that every component operates in sync and performs its assigned tasks at the correct time.

Functions of the Control Unit:

Directs the movement of data between CPU registers, ALU, and memory.

Interprets the operation code (opcode) in the instruction register.

Issues **timing and control signals** to various hardware components.

Coordinates instruction cycle phases: fetch, decode, execute, memory access, and write-back.

Manages branching, jumping, and control transfer instructions.

Types of Control Units:

Hardwired Control Unit:

Uses fixed logic circuits to generate control signals.

Best suited for simple and fast systems.

Microprogrammed Control Unit:

Uses a set of microinstructions stored in a **control memory**.

Ideal for complex instruction sets (like in CISC processors).

4.4.2 Hardwired Control Unit Design and Operation

A **Hardwired Control Unit** uses combinational logic circuits composed of **flip-flops, decoders, multiplexers, and logic gates** to generate control signals. Its behavior is governed by a set of state transitions that occur on clock cycles.

Structure:

Instruction Register (IR): Stores the current instruction.

Decoder: Decodes the opcode and triggers the correct control path.

State Machine (e.g., FSM): Determines control signal sequencing.

Control Signal Generator: Logic circuitry to activate various control lines.

Operation:

Instruction is fetched into the IR.

Decoder interprets opcode bits.

Based on instruction type and current state, the control unit issues signals to:

Select ALU operation.

Enable registers for reading/writing.

Control memory read/write cycles.

The process continues through instruction cycle steps using clock transitions.

Advantages:

High speed: Direct logic paths result in faster execution.

Efficient for RISC architectures with simple instruction sets.

Limitations:

Inflexible: Any change in instruction set requires hardware redesign.

Difficult to modify or expand.

Complex for large instruction sets due to increased logic density.

4.4.3 Microprogrammed Control Organization

A **Microprogrammed Control Unit** generates control signals by executing microinstructions stored in **Control Memory** (usually ROM or RAM). It functions like a simple embedded processor that reads microinstructions and executes them to produce the required control signals.

Structure:

Control Memory (CM):

Stores microprograms.

Each instruction in CM is a **microinstruction**, which corresponds to one micro-operation or a group of operations.

Microinstruction Register (MIR):

Holds the current microinstruction fetched from CM.

Control Address Register (CAR):

Holds the address of the next microinstruction.

Sequencer:

Determines the next address in the control memory based on branching logic or instruction sequence.

Types of Microinstructions:

Horizontal Microinstruction:

Wide word with each bit representing a control signal.

Offers parallelism but consumes more memory.

Vertical Microinstruction:

Encoded fields specify operations (e.g., ALU operation, register selection).

Compact but slower due to decoding requirement.

Operation:

Fetch machine instruction into IR.

Use the opcode to determine the starting address of the corresponding microprogram in control memory.

Sequentially execute microinstructions to perform required operations.

Each microinstruction triggers a specific set of control signals to execute steps such as operand fetch, ALU operation, and result storage.

Advantages:

Easily modifiable: Control logic can be updated by changing microcode.

Supports complex instructions efficiently (ideal for CISC).

Easier to debug and maintain.

Limitations:

Slower than hardwired control due to memory access latency.

Requires more memory and storage for microprograms.

Not well-suited for high-performance RISC systems.

4.4.4 Comparison Between Hardwired and Microprogrammed Control

Feature	Hardwired Control Unit	Microprogrammed Control Unit
Implementation	Fixed logic (gates, flip-flops, FSM)	Microinstructions stored in control memory
Speed	Very fast due to direct signal generation	Slower due to memory access for microinstructions
Flexibility	Low; hardware changes needed for modifications	High; control memory content can be changed easily
Complex Instruction Sets	Hard to implement due to logic complexity	Easy to support via microcode
Design Complexity	High for large ISAs	Easier to design and update
Cost	Low for simple systems	Higher due to memory requirement
Example Use Case	RISC processors	CISC processors, legacy systems

4.5 Pipelining Basics

Pipelining is a fundamental technique used in modern CPUs to enhance **instruction throughput** by overlapping the execution of multiple instructions. Much like an assembly line in manufacturing, pipelining allows different stages of multiple instructions to execute simultaneously, thus increasing the number of instructions processed per unit time. This section explores the concept, implementation stages, performance metrics, and the strengths and constraints of instruction pipelining.

4.5.1 Concept of Instruction Pipelining

Instruction pipelining is a CPU implementation technique where **multiple instructions are divided into smaller subtasks (stages)** that can be executed concurrently. Instead of executing one instruction from start to finish before beginning the next, pipelining allows each instruction to progress through the stages while others enter subsequent stages.

Key Concepts:

Parallelism at the instruction level (ILP – Instruction Level Parallelism).

Each stage completes a specific portion of instruction execution (e.g., fetch, decode, execute).

When the pipeline is full, one instruction can complete **per clock cycle**.

Analogy:

Similar to a car wash, where different steps (soaping, scrubbing, rinsing, drying) occur in parallel on different cars.

4.5.2 Pipeline Stages and Throughput

A typical instruction pipeline consists of 5 classic stages (in RISC architectures):

IF – Instruction Fetch:

Fetches instruction from memory.

ID – Instruction Decode:

Decodes opcode and determines operands.

EX – Execute:

Performs arithmetic or logical operation using ALU.

MEM – Memory Access:

Loads/stores data if required.

WB – Write Back:

Writes result to destination register.

Throughput:

Defined as the number of instructions completed per unit time.

In an **ideal pipeline**, throughput is **one instruction per clock cycle** after the pipeline is filled.

Pipeline Latency:

Time taken for a single instruction to pass through all stages.

Latency does not improve with pipelining, but **overall execution time** reduces due to improved throughput.

4.5.3 Pipeline Performance Metrics

Evaluating the performance of a pipeline involves multiple metrics that quantify speed, efficiency, and resource utilization.

Key Metrics:

- Speedup (S):

$$S = \frac{\text{Execution time without pipeline}}{\text{Execution time with pipeline}}$$

  Ideal speedup = Number of pipeline stages (n), but practical speedup is less due to hazards.

- Efficiency (E):

$$E = \frac{\text{Speedup}}{\text{Number of pipeline stages}}$$

- Throughput (T):

$$T = \frac{\text{Number of completed instructions}}{\text{Total execution time}}$$

- Cycle per Instruction (CPI):
  - For ideal pipelining, CPI ≈ 1
  - Practical CPI > 1 due to stalls and hazards

4.5.4 Advantages and Limitations of Pipelining

Advantages:

Increased Instruction Throughput:

Executes more instructions in less time.

Efficient Resource Utilization:

ALU and other units are used continuously.

Faster Clock Rates:

Shorter sub-operations allow higher clock frequencies.

Modular Design:

Easier to debug and scale each stage independently.

Limitations:

Pipeline Hazards:

Reduce efficiency (data, control, and structural hazards).

Complexity:

Requires sophisticated control logic to handle stalls, flushes, and branching.

Uneven Stage Delays:

Limits the maximum clock speed due to the slowest stage.

Not all instructions can be pipelined equally:

Some multi-cycle operations (e.g., division) don't fit well in a simple pipeline.

4.6 Instruction Hazards

While pipelining improves performance, it introduces **instruction hazards**—situations that prevent the next instruction from executing in the following cycle. Hazards cause **pipeline stalls**, reducing efficiency and requiring advanced control logic to resolve. Understanding these hazards and techniques to mitigate them is critical for effective pipeline design.

4.6.1 Structural Hazards

Structural hazards occur when **hardware resources are insufficient** to handle the current set of instructions in the pipeline.

Causes:

Two instructions simultaneously require the **same resource** (e.g., memory, ALU).

Common in processors that do not have **separate instruction and data memory (Von Neumann architecture)**.

Example:

Both **IF** and **MEM** stages access memory in the same cycle but there's only one memory unit.

Solutions:

Duplicate resources: Use separate instruction and data memory (Harvard architecture).

Resource scheduling: Introduce stalls or design conflict-free execution timing.

4.6.2 Data Hazards

Data hazards arise when **instructions depend on the results of previous instructions**, and the result is not yet available due to pipeline timing.

Types of Data Hazards:

RAW (Read After Write) – True dependency:

Instruction B reads a register that A is writing to.

WAR (Write After Read) – Anti-dependency:

Instruction B writes to a register that A is reading.

WAW (Write After Write) – Output dependency:

Both instructions write to the same register.

Example:

```
ADD R1, R2, R3      ; R1 = R2 + R3
SUB R4, R1, R5      ; Depends on result of ADD
```

Solutions:

Data Forwarding (Bypassing):

Direct result from ALU to next instruction without waiting for WB stage.

Stalling:

Insert no-operation (NOP) instructions to delay dependent instruction.

Register Renaming:

Eliminates false dependencies (WAR and WAW) in superscalar processors.

4.6.3 Control Hazards

Control hazards, also called **branch hazards**, occur when the pipeline makes wrong assumptions about the **flow of control**, typically in branch or jump instructions.

Causes:

The next instruction to fetch depends on the outcome of a **branch**.

Branch resolution occurs in **EX or MEM stage**, but instruction fetch continues assuming no branch.

Example:

```
BEQ R1, R2, LABEL      ; If R1 == R2, branch to LABEL
```

Consequences:

Incorrect instructions are fetched.

Must **flush** or discard these instructions.

Solutions:

Branch Prediction:

Use dynamic/static prediction to guess branch direction.

Delayed Branching:

Reorder instructions so that useful instructions are placed in delay slot.

Speculative Execution:

Execute both paths and discard the wrong one.

4.6.4 Techniques to Minimize Hazards

To minimize pipeline stalls and maximize performance, modern processors employ a range of hardware and compiler-level strategies.

Techniques:

Data Forwarding:

ALU results are forwarded directly to dependent instructions before being written back.

Pipeline Interlocking:

Hardware detects hazards and introduces automatic stalls.

Dynamic Scheduling (e.g., Tomasulo's algorithm):

Out-of-order execution to bypass stalled instructions.

Branch Prediction Units (BPUs):

Predict outcome of branches using history tables.

Register Renaming:

Eliminates false data dependencies by assigning temporary registers dynamically.

Speculative Execution:

Execute instructions beyond branches based on prediction and rollback if incorrect.

Compiler Techniques:

Instruction reordering and loop unrolling to reduce dependency chains.

4.7 Data Hazards

Data hazards are conditions that occur in **instruction pipelines** when instructions that are close together in the pipeline interact in a way that violates **correct program semantics** due to data dependencies. These hazards impact pipeline efficiency and must be resolved through hardware or software techniques.

4.7.1 Read After Write (RAW) Hazards

A **Read After Write (RAW)** hazard, also known as a **true data dependency**, occurs when an instruction depends on the result of a **previous instruction** that has not yet completed.

Example:

```
ADD R1, R2, R3      ; Instruction 1: R1 = R2 + R3
SUB R4, R1, R5      ; Instruction 2: R4 = R1 - R5
```

In this example, instruction 2 needs the result of instruction 1, but R1 is not yet updated when instruction 2 begins execution.

Implications:

If instruction 2 reads R1 before instruction 1 writes it, the result is incorrect.

This hazard is the most common and must be carefully managed in pipelined designs.

4.7.2 Write After Read (WAR) and Write After Write (WAW) Hazards

These hazards are classified as **name dependencies** rather than true data dependencies and usually arise in **out-of-order** execution or **superscalar architectures**.

Write After Read (WAR):

Occurs when an instruction **writes to a register** before a **previous instruction reads** from it.

Example:

READ R1 → Instruction 1 (uses R1)

WRITE R1 → Instruction 2 (writes to R1 before instruction 1 reads it)

WAR hazards are not present in basic in-order pipelines but become critical in advanced execution models.

Write After Write (WAW):

Occurs when two instructions **write to the same register**, and the second write completes **before** the first one.

Example:

WRITE R1 → Instruction 1

WRITE R1 → Instruction 2

Instruction 2's result may overwrite Instruction 1's value if not properly ordered.

4.7.3 Data Forwarding and Hazard Resolution

To maintain instruction throughput, modern processors use **data forwarding (bypassing)** and other hardware techniques to resolve data hazards.

Data Forwarding:

Allows the output of one instruction to be **directly forwarded** to the input of another, bypassing the register write-back stage.

Implemented using **additional data paths and control logic**.

Example:

ALU result from Instruction 1 is forwarded to the ALU input of Instruction 2 in the next cycle.

Other Hazard Resolution Techniques:

Pipeline Stalling (Bubble Insertion):

Temporarily suspends instruction fetch or execution to allow dependencies to resolve.

Register Renaming:

Assigns different physical registers to eliminate WAR and WAW hazards in out-of-order execution.

4.7.4 Compiler-Level Hazard Mitigation

Compilers play a key role in reducing pipeline hazards through **instruction scheduling** and **code reordering**.

Techniques:

Instruction Scheduling:

Reorders instructions to **fill pipeline delay slots** without changing program behavior.

Loop Unrolling:

Duplicates loop bodies to expose more parallelism and reduce control dependencies.

Software Pipelining:

Restructures loops to initiate multiple iterations in overlapping fashion.

Example:

```
; Original (dependent)
ADD R1, R2, R3
SUB R4, R1, R5

; After compiler scheduling (delay filler)
ADD R1, R2, R3
NOP                     ; Delay slot filled
SUB R4, R1, R5
```

These techniques help reduce stalls and maximize pipeline utilization.

4.8 Impact on Instruction Sets

The structure and complexity of **Instruction Set Architecture (ISA)** has a direct impact on **pipeline efficiency** and hazard handling. Over time, ISAs have evolved to better support pipelined execution by minimizing hazards and simplifying control.

4.8.1 Instruction Set Design and Pipeline Compatibility

ISA design influences how instructions are **decoded**, **executed**, and **optimized** for pipelined processors.

Pipeline-Friendly ISA Features:

Fixed-Length Instructions:

Simplifies decoding and speeds up pipeline stages (e.g., RISC).

Few Addressing Modes:

Reduces decode complexity and data hazard frequency.

Simple Operand Access:

Emphasizes register-register operations (load/store architecture).

Importance:

Efficient ISAs ensure **low CPI (Cycles Per Instruction)** in pipelines.

Reduces the occurrence and impact of structural and data hazards.

4.8.2 Influence of Hazards on ISA Development

Instruction hazards have **shaped ISA design**, especially in RISC-based systems that are built for **pipelining efficiency**.

Design Responses to Hazards:

Branch Delay Slots:

Introduced to reduce control hazard penalties.

Separate Load/Store Instructions:

Simplifies data dependencies and memory access.

Reduced Instruction Complexity:

Ensures instructions fit cleanly into pipeline stages.

Impact:

ISAs now favor **regular formats, fewer exceptions**, and **predictable behavior** to align with pipelined execution.

4.8.3 RISC vs. CISC Instruction Set Implications

RISC (Reduced Instruction Set Computer):

Simple, fixed-length instructions.

Emphasizes **pipeline optimization**.

High instruction throughput due to regular instruction patterns.

Easier to decode and schedule in pipeline stages.

CISC (Complex Instruction Set Computer):

Variable-length and multi-cycle instructions.

Requires **microprogrammed control**, complicating pipelining.

Tends to increase pipeline stalls and reduce throughput.

Trade-Off:

RISC is more suited for **deep pipelines and superscalar architectures**.

CISC offers better **code density** but at the cost of execution speed in pipelines.

4.8.4 Pipeline-Oriented Instruction Encoding

To improve pipeline performance, ISAs are designed with **instruction encoding schemes** that facilitate:

Fast decoding

Parallel dispatching

Operand fetching

Techniques:

Fixed field lengths: Opcode, operands, and function codes have defined positions.

Orthogonality: Instruction features are independent and combinable.

Encoding to support register renaming and hazard avoidance.

Example:

VLIW (Very Long Instruction Word) encodes multiple operations in one instruction to enable **explicit parallelism**.

4.9 Parallel Computer Models

As single-core performance reaches physical and thermal limits, modern computing systems employ **parallelism** to enhance performance. Parallel computers use **multiple processing units** that can operate concurrently to solve problems more efficiently.

4.9.1 Concept of Parallel Processing

Parallel processing involves using **two or more processors** (or cores) to execute multiple parts of a task or multiple tasks **simultaneously**.

Types of Parallelism:

Instruction-Level Parallelism (ILP): Pipelining and superscalar execution.

Data Parallelism: Performing the same operation on different data (e.g., SIMD).

Task Parallelism: Different processors execute different tasks concurrently.

Benefits:

Reduced execution time for large-scale computations.

Better utilization of multi-core and multiprocessor systems.

Enables high-performance computing in scientific and engineering applications.

4.9.2 Flynn's Taxonomy: SISD, SIMD, MISD, MIMD

Flynn's taxonomy classifies computer architectures based on the number of **instruction** and **data streams** they process.

Model	Instruction Stream	Data Stream	Description
SISD	Single	Single	Traditional uniprocessor

Model	Instruction Stream	Data Stream	Description
SIMD	Single	Multiple	Vector processing (e.g., GPUs)
MISD	Multiple	Single	Rare; used in fault-tolerant systems
MIMD	Multiple	Multiple	Multiprocessors, clusters, cloud systems

---

4.9.3 Multiprocessor and Multicomputer Architectures

Multiprocessor Systems:

Multiple CPUs sharing **a common memory**.

Tight coupling.

Used in servers, real-time systems.

Multicomputer Systems:

Multiple independent systems connected via network.

Each system has its own CPU and memory.

Loose coupling.

Common in clusters and distributed computing systems.

Types:

Symmetric Multiprocessing (SMP): All CPUs share memory equally.

Asymmetric Multiprocessing (AMP): One CPU controls the others.

4.9.4 Shared Memory vs. Distributed Memory Systems

Shared Memory Systems:

All processors access the **same physical memory**.

Easier to program (global address space).

Requires synchronization to avoid data conflicts.

Performance bottlenecks due to memory contention.

Distributed Memory Systems:

Each processor has **its own local memory**.

Communication is via **message passing**.

Scales better for large systems.

Harder to program due to data partitioning and communication overhead.

4.10 PRAM and VLSI Models

As parallel computing has become essential in high-performance computing and real-time systems, computational models like **PRAM** and **VLSI** are used to evaluate performance, scalability, and architectural trade-offs. The PRAM model provides a

theoretical framework for designing parallel algorithms, while the VLSI model focuses on the hardware cost, chip area, and circuit design constraints in implementing such parallel systems.

4.10.1 Introduction to PRAM (Parallel Random Access Machine) Model

The **Parallel Random Access Machine (PRAM)** is a theoretical model used to **design and analyze parallel algorithms**. It simplifies the complexities of real-world hardware to provide a clean framework for reasoning about **parallelism**, **efficiency**, and **computational complexity**.

Key Characteristics:

- **Multiple processors** execute in parallel.
- All processors share a **common memory**.
- Execution occurs in **synchronous steps** — each step includes reading, computation, and writing.

Importance of PRAM:

- Offers an idealized way to evaluate **parallel algorithms** independently of specific hardware constraints.
- Helps to identify algorithmic **speedup potential**, **work complexity**, and **communication overhead** in parallel systems.
- Used as a foundational model in theoretical computer science, especially in **algorithm analysis and parallel programming**.

Simplifications in PRAM:

- Assumes **unit-time access** to shared memory, regardless of processor count.
- **No contention delay** in memory access, unless explicitly modeled.
- No network latency or processor interconnect constraints.

This abstraction enables researchers to focus on algorithmic design without being hindered by hardware limitations, but its assumptions often require further refinement for practical applications.

4.10.2 PRAM Variants: EREW, CREW, CRCW

To handle real-world issues like **memory access contention**, several PRAM variants are defined based on **concurrent access permissions** to memory locations.

1. EREW (Exclusive Read Exclusive Write):

- **No two processors** can read from or write to the same memory location **simultaneously**.
- Most restrictive and realistic model.
- Simplifies hardware implementation and avoids conflicts entirely.

2. CREW (Concurrent Read Exclusive Write):

Allows **multiple processors to read** from the same location simultaneously.

Only one processor can write to a location at a time.

Common in systems with shared caches or read-broadcast capabilities.

3. CRCW (Concurrent Read Concurrent Write):

Allows **multiple reads and multiple writes** to the same location.

Requires a **conflict resolution policy** for writes:

Common CRCW: All processors must write the **same value**.

Arbitrary CRCW: One processor (randomly or by priority) wins the write.

Priority CRCW: Processor with the highest/lowest ID wins.

Comparison:

PRAM Variant	Concurrent Read	Concurrent Write	Realism
EREW	No	No	High
CREW	Yes	No	Medium
CRCW	Yes	Yes	Low/Theoretical

4.10.3 VLSI Computational Model and Design Consideration

The **VLSI (Very Large Scale Integration)** computational model focuses on designing algorithms and architectures based on **physical constraints of silicon chip design**, such as **chip area**, **wire length**, and **communication cost**.

Key Design Parameters:

Area-Time Complexity (AT complexity):

Measures trade-offs between **chip area (A)** and **computation time (T)**.

AT^2 is used to measure total cost in some VLSI models.

Communication Constraints:

Wire length, routing congestion, and signal delay directly affect performance.

Communication between far-apart units is **costlier** in terms of speed and power.

Gate Delay and Clock Speed:

VLSI designs must account for **propagation delay** and **clock distribution**.

Critical path delay determines maximum clock frequency.

Modularity and Scalability:

Efficient VLSI systems require **modular design**, such as replicated processing units or tiled architectures.

Power Consumption and Heat Dissipation:

As transistor counts increase, **thermal limits** become a critical factor.

VLSI designs must optimize for **low-power operation**, especially in mobile and embedded systems.

Applications of VLSI Model:

Chip-level parallel processing design.

Estimating theoretical performance limits of hardware implementations of algorithms.

Optimizing architecture for matrix multiplication, sorting, or FFT using chip-aware models.

4.10.4 Role of VLSI in Parallel Processing Architectures

VLSI technology plays a **central role in enabling parallel computing** by providing the hardware foundation for integrating **multiple processing units**, **memory blocks**, and **interconnects** on a single chip or multi-chip module.

Contributions of VLSI to Parallel Architectures:

Multi-Core Processors:

VLSI enables the integration of dozens or hundreds of cores on a chip.

Each core can independently execute instructions, supporting **MIMD (Multiple Instruction Multiple Data)** architectures.

System-on-Chip (SoC):

VLSI enables the integration of CPU, GPU, memory, and I/O controllers on a single chip.

Highly parallel components work in coordination for real-time and embedded applications.

Networks-on-Chip (NoC):

Replaces traditional bus systems with **packet-switched interconnects** to improve scalability.

Supports **parallel communication** between multiple cores with minimal latency.

GPUs and Tensor Processing Units (TPUs):

Specialized VLSI designs enable massive **SIMD-style parallelism** for graphics, AI, and scientific computing.

VLSI enables hundreds or thousands of processing units to operate concurrently.

Field-Programmable Gate Arrays (FPGAs):

Reconfigurable VLSI devices used for prototyping and accelerating parallel algorithms.

Useful in **custom parallel data paths** and pipelined computation.

Advantages of VLSI in Parallel Processing:

Compact and energy-efficient hardware.

High **data throughput** due to tightly coupled cores and memories.

Better **latency management** due to shorter interconnects.

Customization for application-specific parallelism (e.g., digital signal processing, cryptography, neural networks).

4.11 Summary

This module on **Advanced Computer Architecture** explores critical concepts that underpin the design and operation of modern processing systems, especially focusing on parallelism, pipelining, control units, and hardware modeling techniques. Beginning with the design of the **Basic Processing Unit (BPU)**, the module introduces the internal architecture of a processor, highlighting the roles of the **control unit**, **data path**, and **register transfer operations**. The interaction between these components forms the foundation for how a processor fetches, decodes, and executes instructions, with emphasis on synchronous operation and control signal generation.

The module proceeds to detail the **instruction execution cycle**, breaking it into key steps: instruction fetch, decode, operand fetch, execution, and result storage. Each stage is tightly coordinated by control signals that synchronize processor activities. As part of processor optimization, **pipelining** is introduced as a technique that enables instruction-level parallelism, thereby improving throughput without increasing clock frequency. The classic five-stage RISC pipeline is discussed, along with performance metrics such as CPI (Cycles Per Instruction), speedup, and efficiency.

With pipelining comes the challenge of **instruction hazards**, which are conditions that cause pipeline stalls or incorrect execution. The module classifies hazards into three categories: **structural**, **data**, and **control hazards**. It presents techniques for resolving or minimizing these hazards, including **data forwarding**, **stalling**, **branch prediction**, and **register renaming**. A deeper look into **data hazards** identifies RAW, WAR, and WAW hazards, which are especially relevant in out-of-order and superscalar execution.

Moving further, the module explores how these concepts influence **Instruction Set Architecture (ISA)**. Pipeline-compatible ISAs adopt simpler, fixed-length instructions with fewer addressing modes to facilitate fast decoding and minimal hazard generation. The distinction between **RISC and CISC** designs is discussed, noting that RISC is more suited to pipelined execution due to its regular instruction patterns and emphasis on load/store architecture.

In the domain of **parallel computing**, the module introduces **Flynn's Taxonomy** (SISD, SIMD, MISD, MIMD) to classify computer systems based on their instruction and data stream handling. It describes **multiprocessor** and **multicomputer** architectures, comparing **shared memory** and **distributed memory** systems in terms of scalability, programming complexity, and communication overhead.

The theoretical framework for parallel algorithm design is introduced through the **PRAM model** (Parallel Random Access Machine), along with its variants (EREW, CREW, CRCW), each modeling different memory access restrictions. The final section focuses on **VLSI (Very Large Scale Integration) computational models**, which consider chip design constraints like area, power, communication delay, and clock speed. The role of VLSI in enabling high-performance parallel architectures such as multicore processors, GPUs, and SoCs is emphasized, tying together theory and practical hardware considerations.

Overall, the module provides a unified view of processor design, pipelining, instruction-level parallelism, and parallel system architectures, integrating both theoretical models and practical design concerns to form a comprehensive understanding of modern computing systems.

4.12 Keywords (One-line Definitions)

Pipelining – A technique that allows overlapping execution of multiple instructions to improve processor throughput.

Data Hazard – A situation where instruction execution depends on the result of a previous, incomplete instruction.

PRAM Model – A theoretical parallel computing model with multiple processors sharing a global memory.

Flynn's Taxonomy – A classification of computer systems based on instruction and data streams (SISD, SIMD, MISD, MIMD).

VLSI Model – A hardware design model focusing on computation cost in terms of area, time, and power in integrated circuits.

Control Unit – The component in a CPU responsible for directing data flow and generating control signals.

4.13 Self-Assessment Questions

Explain the five stages of instruction pipelining in RISC processors and how they contribute to improved throughput.

What are RAW, WAR, and WAW hazards? How do they impact instruction execution in a pipelined processor?

Differentiate between Hardwired and Microprogrammed Control Units with examples of where each is used.

How does instruction set design influence pipeline efficiency? Compare RISC and CISC in this context.

Describe the differences between shared memory and distributed memory in parallel architectures.

What are the key differences between EREW, CREW, and CRCW in the PRAM model?

How does VLSI technology influence the design and scalability of modern parallel computing systems?

4.14 Case Study

Case Study: Designing a Parallel Embedded Processor for Image Processing

An embedded systems company is developing a custom chip for real-time image processing in autonomous drones. The workload requires high throughput, minimal latency, and efficient power usage. The hardware team decides to implement a **RISC-based, pipelined processor** with a **multi-core architecture**. To maximize parallel processing, they incorporate **SIMD instructions** and a **shared memory model** for fast data access. The processor is modeled using **VLSI principles** to optimize layout and reduce interconnect delays. Hazard resolution mechanisms such as **data forwarding** and **branch prediction** are integrated to maintain pipeline efficiency.

Questions Based on Case Study:

Why is a RISC-based pipeline architecture preferred in this scenario, and how does it support real-time image processing?

How do VLSI considerations such as chip area and interconnect delay influence the performance of the embedded processor?

4.15 References

M. Morris Mano & Michael D. Ciletti, *Computer System Architecture*, Pearson Education.

David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.

William Stallings, *Computer Organization and Architecture: Designing for Performance*, Pearson.

Kai Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill.

John P. Hayes, *Computer Architecture and Organization*, McGraw-Hill.

R. V. Damle and N. A. Gokhale, *Parallel Processing*, TechKnowledge Publications.