**Computer Architecture**

**Course Description:**

This course explores how computers are built and how they work. We will cover three main areas:

1. **Instruction Set Architecture (ISA):** is the set of commands that a computer's CPU can understand and execute. It serves as a bridge between software and hardware. Think of ISA as the language of the computer, with its own vocabulary (the instructions) and grammar (the rules for writing those instructions). For example, an ISA might include commands like LOAD, ADD, and STORE, which tell the computer to move data, perform arithmetic, or save results. Each type of computer, such as Intel or ARM, has its own ISA, which means software written for one type may not work on another without modification. ISA is crucial for programmers because it defines how they can communicate with the computer. Understanding ISA allows developers to write efficient programs that can be executed by the hardware. Overall, ISA plays a fundamental role in the design and functionality of computer systems.
2. **Micro-architecture:** When the computer hardware (CPU, memory, buses) receives instructions defined by its **Instruction Set Architecture (ISA)**, the micro-architecture explain how those instructions will be executed by the hardware. It focuses on how different parts of the computer, like the CPU (the brain), memory (where data is stored), and other connections (buses), work together to perform tasks. For example, when the computer needs to add two numbers, micro-architecture decides how to fetch those numbers, perform the addition, and save the result.
3. **System Architecture:** System Architecture is about how all the parts of a computer are organized and how they work together. It looks at the main components, like the CPU (the brain), memory (where data is stored), hard drives (for long-term storage), and devices like keyboards and printers. You can think of it like a city where different buildings (components) are connected by roads (the ways they communicate). For example, when you save a file, the CPU tells the memory where to keep it, and system architecture decides how that information travels between them.

**Course Objectives:**

1. **Discuss Representation of Data and Algorithms:**

We will learn how data is represented in computers (like using 0s and 1s) and how different algorithms (step-by-step instructions) are used to perform tasks on this data. For example, how numbers are stored in memory or how text is encoded.

1. **Demonstrate Different Operations in Terms of Micro-operations:**

**Micro-operations** are the tiny steps that a computer takes to complete a single instruction. When you give the computer a command, it breaks that command down into smaller actions to make it easier to handle. For example, if you want the computer to add two numbers, the steps might include:

1. **Getting the first number** from memory and putting it in a temporary storage area (called a register).
2. **Getting the second number** and putting it in another register.
3. **Adding the two numbers** together using a special part of the computer called the arithmetic logic unit (ALU).
4. **Saving the result** back in memory or another register.

Each of these steps is a micro-operation. By breaking down tasks this way, the computer can manage complex operations more effectively. Understanding micro-operations helps us see how computers perform calculations and process data efficiently. Overall, they are the building blocks that allow computers to carry out larger instructions.

1. **Explain Architecture of Basic Computer and Micro-programmed Control Unit:**
   1. We will look at the basic building blocks of a computer (like the CPU, memory, and input/output systems) and how these parts work together. A micro-programmed control unit is like the manager of these parts, directing them on what to do based on the commands it receives.
2. **Understand Memory and I/O Organization of a Typical Computer System:**

In a typical computer, **memory** is divided into different types, mainly **RAM** (Random Access Memory) and **cache**.

* **RAM** is where the computer keeps the data and programs it is currently using, making it easy for the CPU (the brain of the computer) to access this information quickly.
* **Cache memory** is a smaller and faster type of memory that sits close to the CPU. It stores frequently used data, allowing even faster access when the CPU needs it.

**Input/Output (I/O) devices** are the tools we use to interact with the computer. For example, keyboards allow us to input information, mice help us navigate, and printers output documents. These devices connect to the computer to send and receive data, helping us communicate with the system.

Data moves in and out of the computer through **buses**, which are like pathways that carry information between the CPU, memory, and I/O devices. For instance, when you type on a keyboard, that information goes to the CPU, which processes it and then may send the result to the screen or a printer. Overall, understanding how memory and I/O devices work together helps us see how a computer handles information and interacts with users.

1. **Demonstrate Benefits of Pipelined Systems:**
   1. Pipelining is a technique that allows a computer to work on multiple instructions at the same time, similar to an assembly line in a factory. We will study how this improves performance and speeds up the execution of programs by overlapping different stages of instruction processing.

**Computer Architecture** is the study of how computers are built and work. It covers three main areas:

* **Instruction Set Architecture (ISA):** This is the language computers understand. It defines the commands (instructions) a CPU can execute.
* **Micro-architecture:** This explains how the hardware (CPU, memory, buses) carries out the instructions defined by the ISA.
* **System Architecture:** This focuses on how all the parts of a computer are organized and work together.

**The course objectives include:**

* Understanding how data is represented in computers and how algorithms are used to process it.
* Learning about micro-operations, the small steps a computer takes to execute instructions.
* Exploring the basic components of a computer and how they work together.
* Understanding memory and input/output (I/O) organization.
* Demonstrating the benefits of pipelining, a technique that allows computers to work on multiple instructions at once.

**Real-Life Example of Overlapping Register Windows**

Imagine a restaurant kitchen with multiple chefs working on different dishes. Each chef has their own individual workspace, but they also have a shared pantry where they can store ingredients.

**Traditional Approach:**

* **Separate Workspaces**: Each chef has their own workspace with a limited number of ingredients.
* **Storing and Retrieving**: When a chef needs an ingredient that isn't in their workspace, they have to go to the pantry, retrieve it, bring it back to their workspace, and store it there. This takes time and effort.

**Using a Shared Workspace:**

* **Centralized Storage**: Instead of each chef having their own pantry, there's a central shared pantry.
* **Direct Access**: Chefs can directly access ingredients from the shared pantry without having to store them in their individual workspaces.
* **Reduced Overhead**: This eliminates the extra steps of storing and retrieving ingredients, making the chefs' work more efficient and faster.

**Relating this to Register Windows**

* **Individual Workspaces**: Each register window is like a separate workspace for a function.
* **Shared Pantry**: The overlapping region is the shared pantry between adjacent windows.
* **Efficient Data Sharing**: Data (arguments or return values) can be placed directly in the overlapping region without needing to be saved to memory.
* **Seamless Transition**: When a function is called, the window pointer moves to the next window, but the overlapping region remains accessible to both windows.
* **Avoiding Extra Steps**: By using the overlapping region, functions can directly access and share data without needing to save and restore it in their own private registers. This avoids the overhead of saving and restoring data, making function calls more efficient.

**Efficient Function Calls**

Imagine you're at a restaurant where the chef is cooking multiple courses for a meal. Normally, after finishing one course, the chef would completely clean and reset the kitchen before starting to cook the next course. This would take a lot of time.

Instead, the chef decides to keep some tools and ingredients from the previous course already set up and ready to use. For example, if the knife and cutting board are still needed, they are left in place rather than being put away and taken out again.

By doing this, the chef can start the next dish much faster because everything isn’t being reset completely each time. Similarly, overlapping register windows in a computer allow for efficient switching between tasks without having to completely "clean up" after each one, making the whole process faster.

**Flynn's Classification: A Real-World Analogy**

Imagine a busy restaurant kitchen.

* **Chef** = Instruction stream
* **Tasks** = Data stream

**1. SISD (Single Instruction, Single Data)**

* **Chef**: A single chef preparing a single dish (one dish, one step at a time).
* **Tasks**: The ingredients for that particular dish.
* **Example**: A chef preparing a pasta dish, boiling the pasta, adding sauce, and then grating cheese.

**2. SIMD (Single Instruction, Multiple Data)**

* **Chef**: A single chef using a commercial oven to bake multiple cookies.
* **Tasks**: Multiple items being processed simultaneously with the same instruction.
* **Example**: A pastry chef baking a tray of cookies. All cookies are baked in the same oven at the same time.

**3. MISD (Multiple Instruction, Single Data)**

* **Chefs**: Multiple chefs working on the same dish, each with a different task.
* **Tasks**: A single item being processed by multiple instructions.
* **Example**: A team of chefs preparing a roast. One chef might sear the meat, another might prepare the gravy, and another might roast the vegetables.

**4. MIMD (Multiple Instruction, Multiple Data)**

* **Chefs**: Multiple chefs preparing different dishes, each following their own recipes.
* **Tasks**: Multiple items being processed simultaneously with different instructions.
* **Example**: A busy kitchen with multiple chefs preparing a variety of dishes, each working independently.

In summary:

* **SISD** is like a single chef preparing a single dish.
* **SIMD** is like a chef using a commercial oven to bake multiple cookies at once.
* **MISD** is like multiple chefs working together on a single dish.
* **MIMD** is like a busy kitchen with multiple chefs preparing different dishes.

A diagram of a simd

Description automatically generatedA diagram of a pool

Description automatically generated

**Instruction Pool (green box) =>** These are the program instructions that tell the computer what operations to perform.

**Data Pool (blue box) =>** This contains the data that the instructions will operate on.

**PU (Processing Unit, shown in pink) =>** This is the central component that:

* Fetches instructions from the instruction pool
* Retrieves data from the data pool
* Executes the instructions on the data

**=========================================================**

**Instruction Pool (top):** There is a single instruction stream that broadcasts the same instruction to all Processing Units. This means all PUs will perform the same operation simultaneously.

**Data Pool (left):** Contains different data elements that need to be processed.

**Multiple Processing Units (PUs in pink):** There are multiple processing units (4 shown in this diagram) that:

* All receive the same instruction at once
* Each works on different data elements
* Execute in parallel

**Vector Unit (blue box containing the PUs):** This is the parallel processing component that contains all the PUs working together.

A diagram of a data pool

Description automatically generatedA diagram of a data pool

Description automatically generated

**Instruction Pool (green box):** Contains multiple different instructions. Unlike SISD or SIMD, each Processing Unit receives different instructions to execute.

**Data Pool (blue box):** Contains a single stream of data that is shared between all Processing Units.

**Processing Units (PUs in pink):** Multiple PUs each:

* Receive different instructions
* Work on the same data
* Operate independently but on the same data stream

**=========================================================**

**Instruction Pool (green box):** Contains multiple different instructions. Each Processing Unit can execute different instructions independently.

**Data Pool (blue box):** Contains multiple data streams. Each Processing Unit can work on different data independently.

**Processing Units (PUs in pink):** Multiple PUs (8 shown in this diagram) where each:

* Can execute different instructions
* Can work on different data
* Operates independently of other PUs
* Can coordinate with other PUs when needed

Real-Life Example of Parallel Processing

**Imagine you're building a house.** If you do everything yourself, from laying the foundation to painting the walls, it would take a long time. But if you had a team of workers, each specializing in a different task (like carpenters, electricians, and plumbers), you could finish the house much faster.

**That's essentially what parallel processing does.** It's like having a team of workers (processors or cores) within your computer, each working on a different part of a task at the same time. This makes the task finish much quicker than if just one processor was doing everything.

**Real-Life Example of Parallel Processing: Video Rendering**

**Multi-core processor (parallel processing)**

****

This means the computer has 8 separate central processing units (CPUs) working together to execute tasks. This is called multi-core processor.

**Imagine you're editing a video:** You've added effects, music, and transitions, and now it's time to render the final version. Rendering involves processing each frame of the video, applying the effects and creating the final image.

**With a single-core processor:**

* The computer would process each frame one by one. This could take a long time, especially for high-resolution videos with complex effects.

**With a multi-core processor (parallel processing):**

* The video is divided into smaller sections (perhaps frames or groups of frames).
* Each core of the processor works on a different section.
* For example, one core might handle the effects for a group of frames, while another core handles the audio.
* This significantly speeds up the rendering process, allowing you to create the final video much faster.

This is a common example of parallel processing in action. By dividing the complex task of video rendering into smaller, independent tasks, multi-core processors can achieve significant performance gains.

**Traditional Approach (Without Pipelining)**

* In the traditional, non-pipelined approach, tasks are executed one at a time in a step-by-step manner.
* Each step or stage of a process must be completed before the next one begins. For example, in a CPU instruction execution, the steps would be: **fetch** the instruction, **decode** it, **execute** it, and **write** back the results.
* Only one instruction is processed at a time. Once one instruction is completed, the next one begins.

**Example**

1. Fetch Instruction 1
2. Decode Instruction 1
3. Execute Instruction 1
4. Write Back Instruction 1
5. Fetch Instruction 2  
   …and so on.

**Problem:** This approach results in lower performance because the system has to wait for each instruction to complete before starting the next one, leading to idle stages.

**Pipelining Approach**

* In pipelining, the process is divided into multiple stages, with each stage handling a different part of the task. Each stage operates concurrently, processing different instructions simultaneously.
* As one instruction is being fetched, another can be decoded, and yet another can be executed. This means multiple instructions are being worked on at different stages at the same time.
* This leads to better utilization of the hardware and increased instruction throughput.

**Example**

1. Fetch Instruction 1, while simultaneously decoding Instruction 2, and executing Instruction 3.
2. Continue this process, with different stages working on different instructions concurrently.

### Pipeline Processing

Imagine you're making a sandwich. Instead of doing each step one by one (bread, meat, cheese, etc.), you could have different people working on different parts at the same time. One person could be getting the bread, another could be slicing the meat, and so on. This way, you can make multiple sandwiches faster.

Imagine you’re in a car wash. The car wash has six steps: washing, soaping, scrubbing, rinsing, drying, and polishing. Each of these steps happens in a different section, and the car moves from one section to the next.

**How It Works:**

1. **First Car (Car 1)**: It starts in the first section (washing).
   * While Car 1 is being washed, the other sections are empty.
2. **Second Car (Car 2)**: After Car 1 moves to the next section (soaping), Car 2 enters the first section (washing).
   * Now, two cars are in the car wash, each in a different section.
3. **Third Car (Car 3)**: When Car 1 moves to the third section (scrubbing), Car 2 moves to the second section (soaping), and Car 3 starts in the first section (washing).
4. **Eventually, All Sections Are Busy**: After six cars have entered, every section will be busy with a different car. Now, a new car can enter the car wash every time a car finishes, keeping the whole process moving smoothly.

**What’s Happening Here?**

* **Tasks Are Being Done in Parallel**: Multiple cars are being worked on at the same time, but at different stages.
* **Fast Output**: Once the car wash is filled, a car comes out clean every time a new car enters, making the process very efficient.

In computing, the same idea applies: tasks move through a series of stages in a pipeline, with each stage handling a different part of the work. This allows multiple tasks to be worked on at the same time, just like multiple cars in the car wash.

**Factory Assembly Line Explanation**

* **4 Workstations:** The assembly line has four workstations where tasks are done in sequence: assembling, painting, testing, and packaging.
* **Conveyor Belt Movement:** Items move from left to right along a conveyor belt, stopping at each station for a specific task.
* **Parallel Work:** Once the first item moves to the painting station, a new item can start at the assembling station. As a result, multiple items are being processed at the same time, just at different stages.
* **Continuous Flow:** After an initial setup period, the factory reaches a point where one item comes out fully packaged every time a new item enters the assembling station.

**Technical Pipeline Explanation**

* **4 Segments (S1, S2, S3, S4):** The technical pipeline has four stages where each segment performs a part of the processing task. For example, these stages could be labeled as fetch, decode, execute, and write-back.
* **Data Flow:** Just like the factory's conveyor belt, data moves from one segment to the next, starting from S1 (fetch) and progressing to S4 (write-back).
* **Parallel Execution:** When data moves from one segment to the next, new data can enter the first segment. This means multiple tasks are being processed simultaneously, but at different stages in the pipeline.
* **Pipelined Throughput:** After a few clock cycles (time steps), the pipeline is fully utilized. At this point, a result is produced every clock cycle, similar to how the factory continuously produces items after the line is filled.

**Comparing Both**

1. **Stages of Work:** In both cases, work is broken down into stages or segments, where each stage performs a specific task.
2. **Flow of Items or Data:** Whether it’s items on a conveyor belt or data in a technical pipeline, there is a flow from the first stage to the last.
3. **Parallel Processing:** Both the factory line and the pipeline process multiple tasks simultaneously, with different tasks being at different stages.
4. **Initial Fill Time:** It takes some time before results start coming out continuously, as the pipeline or assembly line fills up with tasks.
5. **Efficiency:** Once fully utilized, both systems operate at maximum efficiency, producing one output per cycle.

The analogy helps understand how pipelines allow multiple tasks to be executed simultaneously by breaking them into smaller, sequential steps, much like an assembly line in a factory.

**Understanding Task Movement in a Pipeline**

A diagram with numbers and letters

Description automatically generated

In a pipeline, each stage processes a specific part of the task. Once a task completes its processing in one stage, it moves to the next stage, allowing the next task to enter the first stage.

Let's break down the example you provided:

**Clock Cycle 2**

* **T1 moves to S2:** This means that task T1 has finished its processing in stage S1 and is now ready for the next stage, S2.
* **T2 enters S1:** Since T1 has vacated S1, the pipeline can now accept a new task. In this case, T2 is the next task in line, so it enters S1 to begin its processing.

**Clock Cycle 3**

* **T1 moves to S3:** T1 has completed its processing in S2 and moves to the next stage, S3.
* **T2 moves to S2:** Similarly, T2 finishes its processing in S1 and moves to S2.
* **T3 enters S1:** As T2 has vacated S1, the pipeline is ready for the next task, which is T3.

**The key point to remember is that**

* Each stage can only process one task at a time.
* Once a task finishes its processing in a stage, it moves to the next stage, making space for the next task.
* This continuous flow of tasks through the pipeline allows for efficient and concurrent processing.

Instruction pipeline

**Segment 1: Instruction Fetch (FI)**

* The computer **fetches** the instruction from memory.
* The fetched instruction is temporarily stored in a FIFO buffer

**Segment 2: Instruction Decode (DA)**

* The fetched instruction is taken from the FIFO buffer and **decoded**.
* The computer determines **what the instruction is asking for**, identifying the type of operation (e.g., addition, subtraction, etc.).
* The operands (data needed for the operation) are identified.
* If the instruction involves **memory addresses**, the **effective memory address is calculated**

**Segment 3: Operand Fetch (FO)**

* The computer **fetches the operand (data)** required for the instruction from **memory or registers**.

**Segment 4: Instruction Execute (EX)**

* The computer **executes** the instruction, means performing the operation specified by the instruction, like (**arithmetic calculations, logical comparisons, or memory access**.)
* Once the operation is complete, the result is stored in the appropriate location (e.g., a register or memory).

**Pipeline Operation (Four segment instruction pipeline)**

**Instruction 1**

* **Starts at Step 1 (IF):** The instruction is fetched from memory.
* **Progresses to Step 2 (DA):** The instruction is decoded, and any necessary addresses are calculated.
* **Moves to Step 3 (FO):** The operands required for execution are fetched.
* **Completes at Step 4 (EX):** The instruction is executed, and the result is prepared for storage.

**Instruction 2**

* **Starts at Step 2 (IF):** While Instruction 1 is in the Decode stage, Instruction 2 is fetched from memory.
* **Progresses to Step 3 (DA):** The fetched instruction is decoded.
* **Moves to Step 4 (FO):** The operands are fetched for this instruction.
* **Completes at Step 5 (EX):** The instruction is executed while Instruction 1 finishes.

**Instruction 3 (Branch)**

* **Starts at Step 3 (IF):** The instruction is fetched while Instruction 2 is being executed.
* **Progresses to Step 4 (DA):** The instruction is decoded, and the effective address is calculated.
* **Moves to Step 5 (FO):** The required operands are fetched.
* **Completes at Step 6 (EX):** The instruction is executed; this may involve branching, affecting the flow of subsequent instructions.

**How is performance of computer increased using pipeline?**

**OR ADVANTAGE**

**Parallel Execution**

* In a pipelined architecture, the execution of instructions is divided into several stages (such as fetching, decoding, executing, and writing back results).
* While one instruction is being executed, another can be decoded, and yet another can be fetched. This overlap means the CPU is always busy and reduces idle time.

### Higher Throughput

Pipelining helps the CPU finish more instructions in a shorter amount of time. In a regular (non-pipelined) CPU, each instruction has to be completed before the next one can start. This means there’s a wait after finishing one instruction.

In a pipelined CPU, however, new instructions can begin before the previous ones are completely finished. So, Increases the number of instructions processed per unit time.

### Improved Clock Rate

Pipelined CPUs can run at higher speeds than non-pipelined ones. This is because the execution process is divided into stages, and each stage can be completed more quickly. Since each instruction goes through these smaller stages, the CPU doesn't need to take as long to complete each one.

As a result, pipelined processors can handle more instructions in a shorter time, which means they can operate at a faster clock rate. This leads to better overall performance and quicker execution of tasks.

What is the Mapping Process?

### Understanding the Mapping Process with a Library Analogy

**Imagine this scenario**

1. **Main Memory as a Library**:
   * Think of the **main memory** (RAM) as a large library filled with many books (data). Each book has a specific address on a shelf in the library.
2. **Cache Memory as a Small Bookshelf**:
   * Now, imagine your **cache memory** as a small bookshelf in your room where you keep your favorite or most frequently read books. It’s much smaller than the library, so it can't hold every book you might need.

### The Process of Getting a Book (Data)

**Step 1: CPU Needs Data**

* When the CPU wants a specific book (data), it generates an address that points to that book's location in the library (main memory).

**Step 2: Check the Bookshelf (Cache)**

* Before going to the library, the CPU checks its small bookshelf (cache) to see if the book is already there.  
  + **If the book is there (cache hit)**
    - The CPU quickly grabs the book from the bookshelf and uses it.
  + **If the book is not there (cache miss)**
    - The CPU has to go to the library to find the book.

**Step 3: Fetching from the Library**

* When the CPU goes to the library, it uses the address to locate the book on the library's shelves. After finding the book, it takes a copy of it back to the small bookshelf (cache) for future reference.

### Understanding the Mapping Process

**The Mapping Process Explained**

* **Mapping** is like deciding where to place books on your small bookshelf. It determines how to organize and store books so you can find them quickly.  
  + **Memory Address**: This is like the title of the book you want. It tells you exactly which book to find in the library.
  + **Cache Address**: This is the specific spot on your small bookshelf where you will place the book once you bring it home from the library.

**Understanding Direct Mapping with a Simple Example**

**Imagine you have a small parking lot**

1. **Parking Lot**: Think of the parking lot as your cache memory.
2. **Parking Spaces**: There are only **4 parking spaces** in this lot. Each space is like a spot in your cache.
3. **Cars**: The cars represent the data you want to store.

**How Direct Mapping Works**

1. **Each Car Has a Number**
   * Each car has a **number** on its license plate, which is like a block number from main memory. For example:
     + Car 1 has the number **1**
     + Car 2 has the number **2**
     + Car 3 has the number **3**
     + Car 4 has the number **4**
     + Car 5 has the number **5**
     + Car 6 has the number **6**
2. **How Cars Park**
   * In direct mapping, each car can only park in a specific space based on its number. To figure out which space a car goes to, you use this simple rule:
     + **Space = (Car Number) mod (Number of Spaces)**
3. **A white background with black text

   Description automatically generatedExample Calculation**:

**What Happens If Another Car Comes?**

Now, let’s say another car with **license plate number 2** tries to park:

A black text on a white background

Description automatically generated

Since Parking Space 2 is already occupied by the car with license plate 6, it will need to move that car to park the new one. In a real cache, this would mean that the data associated with license plate 6 is removed or replaced.

**Direct Mapping**

* Mapping Method

Think of your main memory as a large library and your cache as a small, fast bookshelf.

1. Each book (block of memory) has a fixed spot on the small bookshelf (cache line)
2. Book A always goes on Shelf 2.
3. Book B also happens to always go on Shelf 2.

Associative Mapping:

* Mapping Method

In this library, any book can go on any empty shelf. So if Book A is already on Shelf 2, Book B can go on Shelf 1, 3, or any other available shelf.

Direct Mapping:

Mapping Formula

Simple Formula : Imagine you have a rule that says, "Any book with an even number goes on Shelf 0, and any book with an odd number goes on Shelf 1." This means:

* Book 2 always goes on Shelf 0.
* Book 3 always goes on Shelf 1.

Associative Mapping:

Mapping Formula

**No Fixed Formula**: There’s no rule about which book goes on which shelf. If Shelf 0 is full, you can put a book on Shelf 1, 2, or any available space.

Direct Mapping:

Example

**Specific Shelf for Each Book**: Think of the 4 shelves as having a strict rule about where books go.

* If you bring in Book 6, you use the formula to find its shelf: **6 mod 4 = 2**. So, Book 6 *always* goes on Shelf 2.

Associative Mapping:

Example

**No Strict Rules**: When a new book comes in, it can go on any shelf that’s available. There isn't a specific formula that forces it to go on a particular shelf.

Direct Mapping:

Tag Bits

Think of **tag bits** as the labels you put on the each book. These tag bits help to identify the book and where it should be stored.

**Fixed Shelf**: Each book has a designated shelf where it always goes. For example, if Book 6 always goes on Shelf 2, you don’t need a very detailed label. The system knows that if the label says "Book 6," it should look on Shelf 2.

Associative Mapping:

Tag Bits

**Flexible Shelf**: In this system, books can go on any shelf. There’s no fixed spot, so the label needs to provide more details.

Direct Mapping:

Access Time ((Fast Access)

**You Know Exactly Where to Look**: Because each book has a specific, fixed shelf, you always know where to find it. For example, if Book 6 always goes on Shelf 2, you can go straight to Shelf 2 whenever you need Book 6.

Associative Mapping:

Access Time ((Slower Access)

**You Have to Check All the Shelves**: In this system, any book can be on any shelf. When you look for Book 6, it could be on Shelf 0, 1, 2, or 3—you don’t know exactly which one.

Direct Mapping:

Conflict Misses

 Each locker has a fixed spot where only one specific item can go.

 If two items need the same locker, one gets kicked out, causing a **conflict miss**.

Associative Mapping:

Conflict Misses

 Lockers are more flexible; any item can go in any empty locker.

 Less likely for items to fight over the same space, reducing **conflict misses**.

What do you mean by pipeline? Explain with space time diagram for a six segmented pipeline showing the time it takes to process eight tasks.

Explain an instruction pipeline with an example.

What do you mean by pipelining concept? Discuss various pipeline hazards and their solutions in detail.

**Real-Life Example of Overlapping Register Windows**

*Imagine a restaurant kitchen with multiple chefs working on different dishes. Each chef has their own individual workspace, but they also have a shared pantry where they can store ingredients.*

**Traditional Approach**

* **Separate Workspaces:**  
  Each chef has their own workspace with a limited number of ingredients.
* **Storing and Retrieving:**  
  When a chef needs an ingredient that isn’t in their workspace, they have to:
  + Go to the pantry
  + Retrieve the ingredient
  + Bring it back to their workspace
  + Store it there

This process takes time and effort.

**Using a Shared Workspace**

* **Centralized Storage:**  
  Instead of each chef having their own pantry, there’s a central shared pantry.
* **Direct Access:**  
  Chefs can directly access ingredients from the shared pantry without needing to store them in their individual workspaces.
* **Reduced Overhead:**  
  This eliminates the extra steps of storing and retrieving ingredients, making the chefs’ work more efficient and faster.

**Relating This to Register Windows**

* **Individual Workspaces:**  
  Each register window is like a separate workspace for a function.
* **Shared Pantry:**  
  The overlapping region is the shared pantry between adjacent windows.
* **Direct Data Access:**  
  Data (such as arguments or return values) can be placed directly in the overlapping region without needing to be saved to memory.
* **Efficient Function Calls:**
  + When a function is called, the window pointer moves to the next window, but the overlapping region remains accessible to both windows.
  + By using the overlapping region, functions can directly access and share data without needing to save and restore it in their own private registers.
  + This avoids the overhead of saving and restoring data, making function calls more efficient.

**Common bus system**

### Everyday Scenario: A Restaurant Kitchen

Imagine a restaurant kitchen where chefs are cooking meals. This kitchen can help us understand how a basic computer works.

Key Components

1. **Memory Unit (Pantry)**
   * **What it Does:** Stores all the ingredients (data and instructions) the chefs need.
   * **Capacity:** Think of it like a pantry that can hold a certain number of items (like 4096 ingredients).
   * **How it Works:** Chefs go to the pantry to get ingredients or put them back after using them.
2. **Registers (Kitchen Tools)**
   * **AR (Address Register):** Like a notepad where the chef writes down the location of a specific ingredient in the pantry.
   * **PC (Program Counter):** A checklist that tells the chefs what to do next in the cooking process.
   * **DR (Data Register):** A mixing bowl where ingredients are combined before cooking.
   * **AC (Accumulator):** The main pot where cooking happens, combining everything together (arithmetic and logical operations).
   * **INPR (Input Register):** A scale used to measure ingredients coming from suppliers (input devices).
   * **IR (Instruction Register):** A recipe card that shows what the chef needs to do next (the instruction being executed).
   * **TR (Temporary Register):** A small bowl for holding chopped ingredients while cooking.
   * **OUTR (Output Register):** A serving plate where the finished dish is presented before serving (output devices).
3. **Control Logic (Kitchen Coordinator)**
   * **What it Does:** This is the person in charge who organizes everything in the kitchen. They make sure each task gets done in the right order.
4. **Adder and Logic Circuit (Cooking Methods)**
   * **What it Does:** Represents different cooking methods (like boiling or frying) that combine ingredients in the pot (perform arithmetic and logical operations).

### How It All Works Together

1. **Choosing Tools (Register Selection)**
   * The kitchen coordinator decides which tools (registers) the chefs need for each step. For example, if they need to chop vegetables, they get the knife.
2. **Using Ingredients (Data Loading)**
   * When the chef wants to use an ingredient from the pantry, the coordinator helps them get it. This is like "loading" data from memory.
3. **Fetching Ingredients (Memory Access)**
   * The chef writes down where the ingredient is located in the pantry (address in AR) and asks the coordinator to fetch it.
4. **Following the Recipe (Instruction Execution)**
   * The chef reads the recipe (IR) to know what to do. They gather ingredients, mix them, cook them, and then check what to do next using the checklist (PC).

### Example: Making Soup

1. **Preparation:**
   * The chef looks at the recipe (IR) to see what ingredients they need and where to find them (AR).
2. **Gathering Ingredients:**
   * The coordinator helps the chef measure out the right amounts of vegetables using the scale (INPR).
3. **Cooking:**
   * The chef combines the ingredients in the mixing bowl (DR) and puts them into the main pot (AC) to cook.
   * If the chef needs to figure out cooking times or temperatures, they use the cooking methods (adder and logic circuit).
4. **Serving:**
   * Once the soup is done, the chef places it on a serving plate (OUTR) to serve to customers.

**Memory Unit:**

* Stores data and instructions.
* Has a capacity of 4096 words, each 16 bits wide.
* Connected to the common bus for data transfer.
* Has "write" and "read" control signals to enable data storage and retrieval.

**Registers:**

* **AR (Address Register):** 12 bits, used to store memory addresses.
* **PC (Program Counter):** 12 bits, holds the address of the next instruction to be executed.
* **DR (Data Register):** 16 bits, used for temporary data storage during data transfer.
* **AC (Accumulator):** 16 bits, used for arithmetic and logical operations.
* **INPR (Input Register):** 8 bits, receives data from input devices.
* **IR (Instruction Register):** 16 bits, holds the instruction being executed.
* **TR (Temporary Register):** 16 bits, used for temporary storage.
* **OUTR (Output Register):** 8 bits, sends data to output devices.

**Control Logic:**

* Generates control signals to coordinate the operation of different components.
* Includes logic for instruction decoding, memory addressing, data transfer, and arithmetic/logical operations.

**Adder and Logic Circuit:**

* Performs arithmetic and logical operations on data held in registers.
* Interacts with the common bus and control logic to execute instructions.

**Data Transfer and Control:**

1. **Register Selection:**
   * The binary values of S2, S1, and S0 determine which register's output is connected to the common bus.
   * This allows for flexible data routing between different components.
2. **Data Loading:**
   * A register receives data from the common bus when its "LD" (load) input is activated.
   * This enables selective loading of registers based on control signals.
3. **Memory Access:**
   * The memory unit is accessed by placing the desired address in the AR and activating the "write" or "read" control signals.
   * Data is transferred between the memory and the common bus.
4. **Instruction Execution:**
   * The PC holds the address of the next instruction.
   * The instruction is fetched from memory and loaded into the IR.
   * The control logic decodes the instruction and generates appropriate control signals to execute the instruction.
   * This may involve data transfer between registers, memory access, arithmetic/logical operations, or input/output operations.

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

Imagine two boxes, Box R1 and Box R2. Box R1 has some important items inside it that we want to move to Box R2, but only if a **condition** is met. Let’s say the condition is that a light (P) needs to be on.

In this setup:

* **Box R1** holds the items (like data) we want to transfer.
* **Box R2** is empty, and we want to put the items from R1 into R2, but only when the light is on (P = 1).
* The **clock** represents a time signal that says "now is the right moment for the transfer."

### How This Works in Steps:

1. **Waiting for the Condition**:  
   We’re waiting for the light P to turn on. This is the condition that must be true before we can start transferring the items from Box R1 to Box R2.
2. **Waiting for the Right Moment**:  
   Even if the light (P) is on, we can only move the items when the **clock** tells us the right moment. Think of the clock as a timer that ticks, telling us, "now you can move" at certain intervals.
3. **The Transfer**:  
   When both conditions are met (the light P is on, and the clock says it’s time), we go ahead and transfer the items from Box R1 to Box R2.

Connecting This Back to the Diagram:

**Controlled Transfer Notation (P: R2 ← R1)**:

* This notation means "move the contents of R1 into R2 only when the condition P = 1."
* If P is not 1, no transfer will occur, even if the clock says it's time.
* When the **Load** signal is active and a clock pulse arrives, R2 will accept the data from R1.

**Timing Diagram**:

* The timing diagram shows that there’s a **specific clock pulse** (the one at time t) where the Load signal is active. This is the moment the transfer actually happens.
* So, at time t, with both the rising clock edge and the Load signal active (meaning P = 1), R2 takes in the data from R1.

**In Summary:**

* P = 1 means the condition is met.
* The **clock** tells when to perform the action.
* If both P and the **clock** are ready at the same moment, the data in R1 moves to R2.

**Control Unit Overview**

The **Control Unit** acts like the "brain" of the computer. It manages and coordinates everything that happens inside the CPU by interpreting program instructions and sending signals to other components to carry them out.

**Key Functions of the Control Unit:**

1. **Instruction Fetch**:
   * The CU gets instructions from the computer's memory. This is the first step in the process of running a program.
2. **Instruction Decode**:
   * After fetching, the CU interprets these instructions to understand what action needs to be taken. This step ensures that the CU knows exactly what the instruction is asking for.
3. **Instruction Execute**:
   * Once the instruction is understood, the CU sends signals to other parts of the CPU, like the Arithmetic Logic Unit (ALU), to perform the required actions (like calculations).
4. **Control Flow Management**:
   * The CU also manages the order of execution for instructions. It decides which instruction to run next, based on conditions in the program (like loops or if-statements) and ensures that the CPU follows the correct sequence.

Explain the error detection codes with example.

Draw an instruction cycle state diagram with interrupt and explain it

The question is asking for a diagram that shows the different steps of the CPU’s instruction cycle, including how it handles interrupts that may occur during the cycle. Then, it asks for an explanation of each part of the diagram.

The interrupt cycle allows the processor to pause its current task to handle urgent events, save its state, execute a special routine, and then resume where it left off.

**Interrupt Cycle**

 **Interrupt Request (IRQ)**:  
External devices (like a keyboard, mouse, or timer) or internal events (like errors) send a signal to the CPU, telling it to pause its current task and deal with something important.

 **Interrupt Acknowledgement (INTA)**:  
The CPU checks if interrupts are enabled (IEN = 1), If enabled, then it sends a signal back to the device to acknowledge that it received the interrupt.

 **Save Processor State**:  
Before jumping to handle the interrupt, the CPU saves its current work (including where it was in the program) so it can pick up exactly where it left off later.

 **Branch to ISR (Interrupt Service Routine)**:  
The CPU then jumps to a special piece of code (ISR) that knows how to handle the interrupt, based on the type of interrupt.

 **Execute ISR**:  
The CPU runs the ISR to do whatever needs to be done for the interrupt, like handling an error or processing data from a device.

 **Restore Processor State**:  
Once the interrupt is handled, the CPU restores the saved information (like where it left off in the program).

 **Return to Main Program**:  
Finally, the CPU goes back to the original program, picking up where it was interrupted and continuing as usual.

 **Interrupt Request**: A device or event sends a signal to get the CPU’s attention.

 **Interrupt Acknowledgement**: The CPU checks if it can handle interrupts and acknowledges the request.

 **Save Processor State**: The CPU saves its current task details (like the program's position).

 **Branch to ISR**: The CPU jumps to the interrupt handler code.

 **Execute ISR**: The interrupt handler performs necessary actions.

 **Restore Processor State**: The CPU restores its saved state.

 **Return to Main Program**: The CPU resumes the interrupted program.

Explain the input-output interrupt with example

An input-output (I/O) interrupt is a signal sent to the processor by an I/O device to request immediate attention. This signal interrupts the normal flow of the processor's instruction cycle, allowing it to handle the I/O request efficiently.

A diagram of a computer system

Description automatically generated

**Example: Keyboard Interrupt**

Let's consider a simple example of a keyboard interrupt:

1. **User Presses a Key:** When a user presses a key on the keyboard, the keyboard hardware generates an interrupt signal.
2. **Interrupt Request:** The interrupt signal is sent to the processor.
3. **Interrupt Handling:** The processor saves its current state (program counter, registers, etc.) and switches to an interrupt service routine (ISR) specifically designed to handle keyboard interrupts.
4. **Read Keystroke:** The ISR reads the keycode from the keyboard buffer.
5. **Process Keystroke:** The ISR processes the keystroke, which may involve:
   * Displaying the character on the screen.
   * Adding the character to an input buffer.
   * Executing a specific command based on the key pressed.
6. **Return from Interrupt:** After processing the keystroke, the ISR restores the processor's saved state and returns control to the interrupted program.