**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.
* **Data (arguments or return values)** can be placed directly in the overlapping region without needing to be saved to memory.
* **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)
2. SIMD (Single Instruction, Multiple Data)
3. MISD (Multiple Instruction, Single Data)
4. MIMD (Multiple Instruction, Multiple Data)

**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.

**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.

**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.

**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.

**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 Examples of Pipeline Hazards

**Structural Hazards: A Busy Restaurant**

* **Scenario:** Imagine a small restaurant with a single chef. If multiple customers order complex dishes simultaneously, the chef might be overwhelmed and unable to handle all orders at once.
* **Pipeline Hazard:** The limited resources (the single chef) can't handle the workload (multiple complex orders) simultaneously.
* **Solution 1: Hire more chefs:** Increase the number of chefs to handle multiple orders simultaneously.

**Data Hazards: Waiting for a Friend**

* **Scenario:** You're meeting a friend for lunch. You need to wait for your friend to arrive before ordering.
* **Pipeline Hazard:** The action of ordering (equivalent to an instruction) depends on the result of your friend arriving (a previous event). If your friend hasn't arrived yet, ordering would be premature.
* **Solution 2: Bring your own food:** If you're unsure about your friend's arrival, be prepared with your own food.

**Control Hazards: Changing Plans**

* **Scenario:** You're planning a trip to the beach. If the weather forecast predicts rain, you might decide to change your plans and go to a museum instead.
* **Imagine you're driving a car.** Control hazards are like coming to a fork in the road. You need to decide which way to go based on the signs or your navigation system. If you make the wrong decision, you might have to backtrack or take a detour.
* **Pipeline Hazard:** The decision to change plans (equivalent to a branch instruction) affects the subsequent actions (visiting the museum instead of the beach).
* **Solution 1: Check the weather forecast regularly:** Stay updated on weather conditions to make informed decisions.

Mitigation Techniques

Let's explore real-life scenarios that correspond to the mitigation techniques used to address pipeline hazards:

### Forwarding: Bypassing the Data

* **Scenario:** You're making a sandwich. You need to spread mayonnaise on the bread before adding cheese.
* **Mitigation:** Instead of waiting for the mayonnaise to fully spread, you start adding cheese while the mayonnaise is still being spread. This avoids a delay in the sandwich-making process.

### Branch Prediction: Predicting the Outcome

* **Scenario:** You're driving on a highway and approaching an exit. Based on your previous experience or navigation system, you predict that you'll take the exit.
* **Mitigation:** You start slowing down and preparing to exit even before you reach the exit sign. This avoids sudden braking or lane changes if your prediction is correct.

### Delayed Branches: Inserting a Nop

* **Scenario:** You're planning a trip and need to decide whether to go to the beach or the mountains. You know that checking the weather forecast will take a few minutes.
* **Mitigation:** You decide to wait a few minutes before making a decision to ensure you have the most accurate weather information.

### Pipeline Scheduling: Rearranging Instructions

* **Scenario:** You're assembling a piece of furniture. You realize that you can install the legs before attaching the top panel, which will save time.
* **Mitigation:** You rearrange the assembly steps to optimize the process and avoid unnecessary delays.

Understanding the Store-Program Concept: A Real-Life Analogy

**Imagine a cookbook.**

* **Instructions:** The recipes in the cookbook are like instructions for a computer.
* **Memory:** The cookbook itself is like the computer's memory. It stores the instructions (recipes).
* **Program Counter:** As the cook, you follow the steps in the recipe in order. Similarly, in a computer, a special register called the program counter keeps track of which instruction to execute next.
* **Fetch-Execute Cycle:** When you cook, you:
  + - **Fetch:** Look at the next step in the recipe.
    - **Decode:** Understand what the step means (e.g., "add 1 cup of flour").
    - **Execute:** Perform the action (add the flour).
    - **Increment PC:** Once you've completed a step, you move on to the next one.

**Real-Life Example:**

Let's say you want to bake a cake.

1. **Instructions:** You open your cookbook to the cake recipe.
2. **Fetch:** You read the first step: "Preheat oven to 350°F."
3. **Decode:** You understand that you need to set the oven temperature.
4. **Execute:** You go to the oven and adjust the temperature.
5. **Increment PC:** You move on to the next step in the recipe.

**In essence, a computer's store-program concept is like following a recipe, where the recipe is stored in memory and the computer follows the instructions step by step.** This flexibility allows computers to perform a wide range of tasks, from simple calculations to complex simulations, by simply changing the instructions (recipes) they are given.