ECE 460 - GPU Architecture

Week 2

Let's do a quick refresher on the MIPS 32-bit processor you designed, focusing on the key concepts of its instruction set architecture (ISA) and pipelined datapath. This will serve as a familiar starting point. Then, we'll pivot to the core of today's session: an introduction to GPU architecture. We'll explore the fundamental differences between CPUs and GPUs, and I'll introduce a very simple GPU design with a minimal ISA. To make it tangible, we'll walk through a basic program and see how it would execute on our simple GPU.

By the end of this session, you should have a solid conceptual understanding of the architectural principles that make GPUs so powerful for certain tasks. We'll be keeping things at a high level today, focusing on the theory and design. In the coming weeks, we'll dive deeper into the SystemVerilog implementation of this simple GPU.

Part 0: Flynn's Notation - A framework for understanding the processor architectures

In the world of computer architecture, we need a way to classify different types of computer systems based on how they handle instructions and data. In 1966, Michael J. Flynn developed a simple yet powerful classification system known as **Flynn's Taxonomy** (or Notation). This taxonomy categorizes computer architectures based on the number of concurrent instruction streams and data streams available in the system.

Think of it as a high-level way to group processors by their parallel computing capabilities. It focuses on two fundamental concepts:

1. `Instruction Stream`: The sequence of instructions being executed by the processor.
2. `Data Stream`: The sequence of data being manipulated by those instructions.

By looking at whether these streams are single or multiple, Flynn created a 2x2 matrix that gives us four distinct classifications.

The Four Classifications

Here is a brief explanation of each of the four categories in Flynn's Notation.

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

- **Description**: This is the classic, non-parallel computer architecture. It features a single control unit that fetches and executes a single stream of instructions, which operate on a single stream of data. One instruction is processed at a time on one set of data.
- **Analogy**: Think of a single chef following one recipe (one instruction stream) to cook one dish (one data stream).
- **Example**: The MIPS processor you designed in ECE 251 is a perfect example of a SISD architecture. Early personal computers and most simple microcontrollers also fall into this category.

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

- **Description**: This architecture features a single control unit that broadcasts a single instruction to multiple independent processing elements (PEs). Each PE executes that same instruction simultaneously, but on its own unique piece of data. It's a model built for data-level parallelism.
- **Analogy**: Imagine a drill sergeant (single instruction) telling an entire platoon of soldiers (multiple data streams) to "do ten push-ups." Everyone does the same thing at the same time, but on their own body.
- **Example**: This is the model for our simple GPU. Graphics processors are the quintessential example of SIMD, as they often apply the same operation (e.g., shading a pixel) to massive amounts of data (all the pixels in a polygon). CPU instruction set extensions like MMX, SSE, and AVX also use SIMD principles.

3. MISD (Multiple Instruction, Single Data)

- **Description**: In this model, multiple instruction streams operate in parallel on a single data stream. Several processing units receive different instructions but work on the same data.
- **Analogy**: Imagine a single assembly line (single data stream) where multiple quality control inspectors (multiple instruction streams) are performing different checks on each item as it passes by.
- **Example**: This is the rarest of the four classifications and has seen very few commercial implementations. It is sometimes used in fault-tolerant systems where multiple redundant processors perform different calculations on the same input to verify the correctness of the result.

4. MIMD (Multiple Instruction, Multiple Data)

- **Description**: This is the most flexible and powerful form of parallel computing. A MIMD system contains multiple, independent processors, each executing a different instruction stream on its own separate data stream. The processors can work on completely unrelated tasks or coordinate to solve a larger problem.
- **Analogy**: Think of a large commercial kitchen with several chefs (multiple instruction streams), each working on a different recipe with their own ingredients (multiple data streams) to create a complete banquet.
- **Example**: Modern multi-core CPUs are the most common example of MIMD architecture. Each core can run a different program or a different thread of the same program independently. Supercomputers and distributed computing clusters also fall into this category.

Understanding this taxonomy helps us place our designs in the proper context. Our MIPS CPU was SISD, and the GPU we are now designing is a classic example of SIMD. This distinction is fundamental to why GPUs are so effective at graphics and scientific computing, while CPUs excel at general-purpose, sequential tasks.

Part 1: MIPS 32-bit CPU Review

Let's begin by jogging our memory about the MIPS 32-bit processor you masterfully designed in ECE 251. The MIPS architecture is a classic example of a Reduced Instruction Set Computer (RISC), characterized by a small, highly optimized set of instructions.

Instruction Set Architecture (ISA):

The MIPS ISA is a **load-store architecture**, meaning that only load and store instructions can access memory. All other operations, like arithmetic and logical instructions, operate on registers. This design choice simplifies the control logic and allows for a more efficient pipeline. You'll recall the three main instruction formats: R-type for register-to-register operations, I-type for immediate and data transfer instructions, and J-type for jumps.

Single-Cycle vs. Pipelined Datapath:

In ECE 251, you designed a single-cycle processor. While conceptually simple, where each instruction is executed in a single clock cycle, it's not very efficient. The clock cycle time is determined by the longest instruction, which slows down the entire processor.

To improve performance, you then moved on to a pipelined MIPS processor. By breaking down the instruction execution into five stages—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)—we can overlap the execution of multiple instructions. This is analogous to an assembly line, where different stages work on different parts of the production process simultaneously, significantly increasing the instruction throughput. The key to this is the use of pipeline registers between each stage to hold the intermediate results.

The following table illustrates the 5-stage MIPS pipeline.
Classic 5-Stage MIPS Pipeline:

1. `IF`: Instruction Fetch
2. `ID`: Instruction Decode & Register Read
3. `EX`: Execute or Address Calculation
4. `MEM`: Memory Access
5. `WB`: Write Back

Cycle	1st Instr	2nd Instr	3rd Instr	4th Instr	5th Instr
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB	MEM	EX	ID	IF
6		WB	MEM	EX	ID
7			WB	MEM	EX
8				WB	MEM
9					WB

This concept of pipelining is a form of instruction-level parallelism. It's a crucial technique for speeding up sequential programs. However, as we'll see, GPUs take parallelism to a whole new level.

Part 2: Introduction to a Simple GPU Architecture

While CPUs are optimized for low-latency execution of a single instruction stream, GPUs are designed for high-throughput, parallel processing of massive amounts of data. Think of a CPU as a sports car, designed for speed on a single task, while a GPU is like a fleet of buses, designed to move a lot of people (data) at once.

The key architectural difference lies in the number of processing cores. A CPU typically has a few powerful cores, whereas a GPU has hundreds or even thousands of smaller, more specialized cores. This massive parallelism is what makes GPUs so effective for tasks like graphics rendering, scientific computing, and machine learning.

To understand how this works, we're going to design a very simple GPU. Our goal here is not to build a commercial-grade GPU, but to grasp the core concepts.

A Minimal GPU ISA:

Let's define a very simple Instruction Set Architecture for our GPU. We'll keep it minimal to illustrate the key ideas.

Instruction	Opcode	Description
`LOAD Rx, [Ry]`	`0001`	Load data from memory address in Ry into register Rx.
`STORE Rx, [Ry]`	`0010`	Store data from register Rx to memory address in Ry.
`ADD Rx, Ry, Rz`	`0011`	Add the values in Ry and Rz and store the result in Rx.
`HALT`	`1111`	Stop execution.

This is a very basic set of instructions, but it's enough to perform a simple but common parallel task: **vector addition**.

A Simple SIMD GPU Architecture:

Our simple GPU will be a SIMD (Single Instruction, Multiple Data) machine. This means that a single instruction is executed simultaneously by multiple processing elements (our "cores") on different data.

Here's the high-level architecture of our simple GPU:

- **A small number of cores:** Let's say we have 4 simple cores. Each core has its own small set of registers (e.g., `R0` - `R3`).

- **Shared Instruction Memory:** All cores will fetch instructions from the same instruction memory. This is where the **Single Instruction** part of SIMD comes from.
- **Shared Data Memory:** All cores can access a shared data memory.
- **A simple Control Unit:** This unit will fetch the instructions and broadcast them to all the cores.

```
+---------------------------------------------------+
|                  Control Unit                     |
+---------------------------+-----------------------+
|   Shared Instruction Mem  | Shared Data Memory    |
+---------------------------+-----------------------+
|   +----------+   +----------+   +----------+      |
|   |  Core 1 |   |  Core 2 |...|  Core N |       |
|   +----------+   +----------+   +----------+      |
|   |Regs |ALU|   |Regs |ALU|   |Regs |ALU|       |
|   +----------+   +----------+   +----------+      |
+---------------------------------------------------+
```

- `Regs` are per-core registers; `ALU` is the per-core arithmetic-logic unit.
- All cores receive instructions from shared instruction memory and share access to shared data memory.
- The control unit oversees this architecture by dispatching instructions and managing coordination among cores.

This structure enables high parallelism by allowing many computations to run simultaneously with minimal control and maximal data throughput.

Example Program: Vector Addition

Now, let's see how we can use our simple GPU to perform vector addition. Imagine we have two vectors, A and B, and we want to compute their sum and store it in a vector C.

Let's say our vectors have a length of 4.

```
A = [A0, A1, A2, A3]
B = [B0, B1, B2, B3]
C = [C0, C1, C2, C3]
```

In memory, we'll lay out the data like this:

```
Address 0-3: Vector A
Address 4-7: Vector B
Address 8-11: Vector C (for the results)
```

Here is a **simple program** in our GPU's ISA to perform this vector addition:

```
// Each core will execute this program in parallel on a different element of the
vectors.
// For simplicity, we'll assume each core has a unique ID (0-3) that it can use
// to calculate its memory offsets.  In a real GPU, this is handled by the
hardware.

// Core 0 will process A[0] and B[0], Core 1 will process A[1] and B[1], and so
on.

LOAD R1, [R0 + 0]    // Load A[i] into R1 (R0 holds the base address for A for
this core)
LOAD R2, [R0 + 4]    // Load B[i] into R2 (offset of 4 for vector B)
ADD R3, R1, R2       // C[i] = A[i] + B[i]
STORE [R0 + 8], R3   // Store the result in C[i] (offset of 8 for vector C)
HALT
```

Execution Walkthrough:

Now, let's trace the execution of this program on our 4-core GPU. The control unit fetches the first instruction, `LOAD R1, [R0 + 0]`, and broadcasts it to all four cores.

- `Core 0:` `R0` is initialized to `0`. It loads the value at memory address `0` (`A0`) into its `R1`.
- `Core 1:` `R0` is initialized to `1`. It loads the value at memory address `1` (`A1`) into its `R1`.
- `Core 2:` `R0` is initialized to `2`. It loads the value at memory address `2` (`A2`) into its `R1`.
- `Core 3:` `R0` is initialized to `3`. It loads the value at memory address `3` (`A3`) into its `R1`.

In a single cycle (conceptually), all four cores have loaded their respective elements from vector A.

The control unit then fetches and broadcasts the next instruction, `LOAD R2, [R0 + 4]`.

- `Core 0:` Loads the value at memory address `4` (`B0`) into its `R2`.

- `Core 1:` Loads the value at memory address `5` (`B1`) into its `R2`.
- `Core 2:` Loads the value at memory address `6` (`B2`) into its `R2`.
- `Core 3:` Loads the value at memory address `7` (`B3`) into its `R2`.

Next, the `ADD` instruction is executed in parallel on all cores, and finally, the `STORE` instruction writes the results back to the corresponding locations in `vector C`.

As you can see, in just a few instruction cycles, we have performed four additions. A single-core CPU would have had to loop through these operations four times. This is the power of data parallelism. Believe! :)

From Theory to SystemVerilog:

Next, we will start translating this conceptual design into a SystemVerilog implementation. We'll design the individual modules: the core, the memory interfaces, and the control unit. You'll find that many of the skills you learned in designing the MIPS processor, such as creating ALUs, register files, and control logic, will be directly applicable here. The main new challenge will be in orchestrating the parallel execution across multiple cores.