1 Introduction

1.1 What's a Tensor Core and Why Should You Care?

You've probably heard a lot about GPUs powering artificial intelligence. At the heart of these GPUs are specialized hardware blocks known as **tensor cores**. Their sole purpose is to crunch through one specific operation blazing fast:

$$E = A \times B + D \tag{1}$$

where A, B, D, and E are *matrices*. This one operation—multiplying two matrices and adding a third—happens billions of times during the training and running of AI models.

The good news? A tensor core isn't magic. It's built from the exact same fundamental components you've been learning about in class: **adders, multipliers, registers, multiplexers**, and **finite state machines**. In this project, you're going to prove that to yourself by building a simple one from scratch.

1.2 Prerequisites

Before diving in, make sure you're comfortable with:

- **Combinational circuits**: Logic gates (AND, OR, XOR, NOT), half adders, full adders, multiplexers, and decoders.
- **Sequential circuits**: Flip-flops (D flip-flops), registers, counters, shift registers, and basic finite state machines (FSMs).
- **Verilog basics**: Module declarations, `wire` vs. `reg`, `assign` statements, `always` blocks, module instantiation, and writing basic testbenches.

If any of these feel a bit rusty, take some time to review the TA materials and your past Verilog homework before getting started.

1.3 Project Overview

We've broken down the design of the tensor core into four phases. Each phase builds directly on the previous one:

1. **Phase 1 — Adders and a Multiplier** (combinational): Build the small arithmetic circuits that will serve as your core building blocks.
2. **Phase 2 — Matrix Multiplier with a Pipeline** (combinational + sequential): Connect your Phase 1 circuits together and add registers to create a pipelined architecture.
3. **Phase 3 — Tensor Module** (combinational): Add a matrix adder to complete the full equation $A \times B + D = E$.
4. **Final Phase — Control and Multi-Precision** (sequential): Build an FSM to control the entire system and allow it to handle different data precisions.

1.4 Learning Goals

By the end of this project, you should be able to:

- Build a complex system by combining smaller, tested modules (hierarchical design).
- Understand how registers are used to create pipelines that process data in stages.
- Design an FSM to control a datapath.
- Write robust, self-checking testbenches.
- See how the basic circuits we cover in class are actually used in real-world processors.

2 Background Concepts

Let's quickly review the concepts you'll need for each phase. It's a good idea to read through this section before you start writing any Verilog.

2.1 Review: How Addition Works in Hardware

You've already built a **full adder** in your homework. It takes three 1-bit inputs (A, B, C_{in}) and produces a 1-bit sum S and a carry-out C_{out}:

$$S = A \oplus B \oplus C_{in} \tag{2}$$

$$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B)) \tag{3}$$

To add two 4-bit numbers, you just chain four of these full adders together so the carry-out of bit 0 feeds the carry-in of bit 1, and so on. We call this a **ripple-carry adder** because the carry literally "ripples" through the chain.

Example: Let's add 5_{10} (`0101`) and 6_{10} (`0110`):

- **Bit 0 (rightmost):** $1 + 0 = 1$ (Sum: 1, Carry: 0)
- **Bit 1:** $0 + 1 +$ (Carry 0) $= 1$ (Sum: 1, Carry: 0)
- **Bit 2:** $1 + 1 +$ (Carry 0) $= 0$ (Sum: 0, Carry: 1)
- **Bit 3 (leftmost):** $0 + 0 +$ (Carry 1) $= 1$ (Sum: 1, Carry: 0)

The final result is `1011` (11_{10}). Notice how Bit 3 had to wait for the carry from Bit 2.

The downside of ripple-carry is that bit 3 can't finish until bit 2 is done, which relies on bit 1, and so on. A **carry-lookahead adder** speeds this up by calculating the carries all at once using extra logic. You don't need a full carry-lookahead implementation for this project; as long as your carry adder correctly handles the carry-in and carry-out, you'll be fine.

2.2 Review: How Multiplication Works in Hardware

Think about how you do long multiplication on paper. To compute 13×11 in binary (1101×1011):

1. Write down a **partial product** for each bit of the second number.
2. Shift each partial product to the left based on its bit position.
3. Add all those partial products together.

Example in Hardware:

```
      1101   (Multiplicand: 13)
    x 1011   (Multiplier: 11)
    ------
      1101   (1101 AND 1, Shifted 0 bits)
     11010   (1101 AND 1, Shifted 1 bit)
    000000   (1101 AND 0, Shifted 2 bits)
+  1101000   (1101 AND 1, Shifted 3 bits)
----------
  10001111   (Result: 143)
```

Notice that each partial product is just the first number ANDed with a single bit from the second number. Shifting in hardware is practically free (it's literally just how you wire the signals). The addition is handled by your adders. So, a 4-bit multiplier is essentially just **AND gates plus adders**. This is exactly what you'll be building in Phase 1.

2.3 What Is a Matrix and Matrix Multiplication?

A **matrix** is just a grid of numbers. For this project, we'll stick to 4×4 matrices (4 rows, 4 columns). Here's an example:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

When you multiply two matrices $C = A \times B$, each element of the result is calculated like this:

$$C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j} + A_{i3} \cdot B_{3j} + A_{i4} \cdot B_{4j} \tag{4}$$

In plain English: you take row i of A, take column j of B, multiply the matching elements, and add them all up. This is known as a **dot product**.

For a single 4×4 matrix multiplication, you'll need to do:

- 64 multiplications (4 per element $\times$ 16 elements).
- 48 additions (3 additions to sum 4 products, for each of the 16 elements).

2.4 What Is a Pipeline? (Registers Between Stages)

You already know that a **register** saves a value on the rising edge of the clock. A **pipeline** is simply what happens when you insert registers between combinational stages so that different parts of your circuit can work on different pieces of data simultaneously.

How to think about it: Imagine grading 100 exams. If you do it all yourself—reading, grading, and recording the score—it takes forever. But if you form an assembly line with two TAs, the process speeds up. You read the answers and pass the paper to the first TA, who grades it and passes it to the second TA to record. Once the line is full, a graded exam pops out at the end *every time you hand off a new paper*. The overall throughput skyrockets, even though each exam still goes through all three steps.

The classic analogy is a car wash: soap, rinse, dry. Without pipelining, a car has to finish the entire wash before the next one enters. With pipelining, three cars are inside at the same time.

In hardware, combinational logic (like your adders) takes time to settle. If you chain too many gates together, the critical path gets too long, and you're forced to drop your clock speed. By inserting **registers** (checkpoints) in the middle, you break that long path into shorter, faster stages. In Phase 2, you'll use registers to pipeline your matrix multiplier so you can feed it new data every clock cycle without waiting for the last calculation to finish.

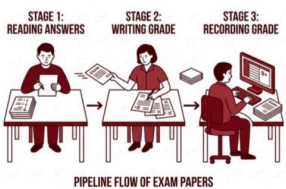

Figure 1: Illustration of a pipeline using the exam grading assembly line analogy.

2.5 What Is a Systolic Array?

A **systolic array** is a grid of identical, small processing cells. Data pulses through the grid rhythmically, kind of like a heartbeat (which is where the word "systolic" comes from). Each cell does one quick multiply-and-add, then hands the data off to its neighbor.

How to think about it: Picture a classroom with 16 students in a 4×4 grid. You hand a number to the first student in each row on the left, and another number to the first student in each column at the front. Every time you clap (the clock tick), each student multiplies their two numbers, adds the result to their running total, and then *passes the incoming numbers* to the student on their right and the student behind them. Instead of everyone getting up to check a central blackboard (memory) over and over, the numbers just flow rhythmically through the room.

For our 4×4 systolic array:

- There are 16 cells arranged in a 4×4 grid.
- Row data from matrix A slides in from the **left**.

- Column data from matrix B slides in from the **top**.
- Each cell multiplies the values it receives, adds the product to a running sum, and passes the original values along.

The real magic here is **data reuse**. Each value from A or B is used by multiple cells as it travels through the grid, saving you from having to fetch it from memory multiple times.

Think of it like a conveyor belt: each cell grabs a piece of data as it passes by, does a tiny bit of math, and lets the data keep moving.÷

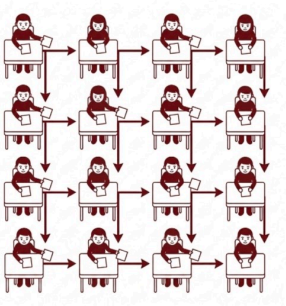

Figure 2: Illustration of a 4×4 systolic array using the classroom analogy.

2.6 What Is a Finite State Machine (FSM)?

You've designed FSMs in class before.

How to think about it: The circuits you build in Phases 1–3 (like your adders and multipliers) make up the datapath. They can do the math, but they don't actually know *when* to start, *when* to stop, or how to handle different data sizes. That's where the Finite State Machine comes in. The FSM acts as the "brain" or control unit. It looks at the current state of the system and any incoming signals (like a `start` button) and tells the datapath exactly what to do and when to do it.

An FSM consists of:

- A set of **states** (e.g., IDLE, LOAD, COMPUTE, OUTPUT).
- **Transitions** between states triggered by input signals or conditions (like a stoplight changing from red to green).
- **Outputs** that depend on the current state (Moore machine) or the state plus inputs (Mealy machine).

In the Final Phase, you'll design an FSM that manages your tensor core, telling it when to load in data, when to calculate, and when to spit out the final results.

3 Project Phases

3.1 Phase 1: Adders and Multiplier

3.1.1 What You're Building

Three completely combinational modules. No clocks, no registers, no `always @(posedge clk)`. These are the raw building blocks you'll use for everything else.

3.1.2 Modules to Implement

Module Name	What It Does
`adder_4bit`	Adds two 4-bit numbers with a carry-in. Outputs a 4-bit sum and a carry-out. Use four full adders chained together (ripple-carry).
`carry_adder_4bit`	A faster version of the 4-bit adder that handles carry-in and carry-out properly. You can go with a carry-lookahead or carry-select approach.
`multiplier_4bit`	Multiplies two 4-bit numbers to get an 8-bit result. **Important:** You have to build this from AND gates and your adder modules. Do **not** just use the Verilog `*` operator!

3.1.3 Hints

- For the multiplier, generate the 4 partial products using AND gates, then sum them up using your adders.
- Partial product 0 has no shift. Partial product 1 is shifted left by 1 bit, partial product 2 by 2 bits, and so on.
- Use structural Verilog (module instantiation) to connect your adders inside the multiplier.

3.2 Phase 2: Matrix Multiplier and Systolic Array

3.2.1 What You're Building

Now you'll use your Phase 1 modules to build circuits that multiply two 4×4 matrices. This is where **registers** (sequential logic) make their first appearance.

3.2.2 Modules to Implement

Module Name	What It Does
`matrix_mult_4x4`	Multiplies two 4×4 matrices. Inside, you'll instantiate 16 multiply-accumulate units (each using your `multiplier_4bit` and `adder_4bit`).
`mem_reuse_pipe`	A set of **D flip-flop registers** that delay the input data so that each value reaches the right processing cell at the exact right clock cycle. Think of them as shift registers of varying lengths.
`systolic_4x4`	The full 4×4 systolic array. It hooks up the matrix multiplier to the memory reuse pipeline. Inputs: clock, reset, two matrices. Output: one result matrix.

3.2.3 Hints

- **Processing Element (PE)**: Start by designing just one PE. A PE takes `a_in`, `b_in`, and `sum_in`. It calculates `sum_out = sum_in + (a_in × b_in)` and passes `a_in` and `b_in` straight through to its neighbors. Use a register to hold the running sum.
- **Staggering**: Matrix A's rows and matrix B's columns need to be staggered (delayed) so they meet at the right cell at the right time. Row 0 enters at clock cycle 0, row 1 at cycle 1, row 2 at cycle 2, etc. Use shift registers (chains of D flip-flops) to create these delays.
- **Clock and Reset**: Make sure every register has a synchronous reset. When `reset` is high, all those running sums should clear back to zero.

3.3 Phase 3: Tensor Module

3.3.1 What You're Building

A matrix adder and the final tensor module. The tensor module computes the complete equation $E = A \times B + D$.

3.3.2 Modules to Implement

Module Name	What It Does
`matrix_add_4x4`	Adds two 4×4 matrices element by element. Inside, you'll instantiate 16 copies of your `adder_4bit`. This is fully combinational.
`tensor_module`	Connects your `systolic_4x4` output to the `matrix_add_4x4`. It takes three input matrices (A, B, D) and gives you the output matrix $E = A \times B + D$.

3.3.3 Hints

- The matrix adder is pretty straightforward: just 16 adders running in parallel, one for each element.

- For the tensor module, route the output of the systolic array ($C = A \times B$) straight into one side of the matrix adder, and matrix D into the other.

3.4 Final Phase: Control Unit and Multi-Precision

3.4.1 What You're Building

An FSM-based control unit that acts as the brains of the operation, allowing your tensor core to process 8-bit, 4-bit, or 2-bit numbers.

3.4.2 Modules to Implement

Module Name	What It Does
`data_ctrl_unit`	An FSM that reads a 2-bit mode input (`00` = 2-bit, `01` = 4-bit, `10` = 8-bit) and configures the datapath. It uses multiplexers to pick the correct data width.
`io_manager`	Handles inputs and outputs using registers. It includes a `valid` output signal that flips high when the result is ready, and a `start` input signal that kicks off the computation.
`simple_tensor_core`	The top-level module. It wires together the `tensor_module`, `data_ctrl_unit`, and `io_manager`. This is your final deliverable.

3.4.3 Multi-Precision: What It Means

Supporting multiple precisions just means your hardware can handle different number sizes dynamically:

- In **8-bit mode**, each matrix element is an 8-bit number (from 0–255).
- In **4-bit mode**, each element is a 4-bit number (from 0–15). You can pack two 4-bit numbers into a single 8-bit register.
- In **2-bit mode**, each element is a 2-bit number (from 0–3). You can pack four 2-bit numbers into one 8-bit register.

Use **multiplexers** to select which bits of the input registers actually get sent to the multiplier depending on the current mode. It's the same MUX logic you've seen in class.

4 Grading Criteria

Criterion	Weight	What We Look For
Functional Correctness	70%	
Report and Documentation	25%	
Code Readability	5%	

5 Tips for Success

1. **Build bottom-up**: Start with Phase 1 and don't move on until every testbench passes. A tiny bug in your adder will break everything above it.
2. **Test as you go**: Write your testbench *before* or *while* you write the module. This is called test-driven development, and it will save you hours of debugging.
3. **Draw before you code**: Sketch out your block diagrams and signal connections on paper before you type any Verilog. It really helps prevent silly wiring mistakes.
4. **Use meaningful names**: `a_in`, `b_in`, and `sum_out` are vastly easier to debug than `x`, `y`, and `z`.
5. **Simulate often**: Run your testbench after every small change. Don't write 200 lines of code and then simulate it all at once for the first time.
6. **Ask for help early**: If you're stuck on Phase 1 for more than a couple of days, ask your instructor or TA. The rest of the project depends on getting the foundation right.

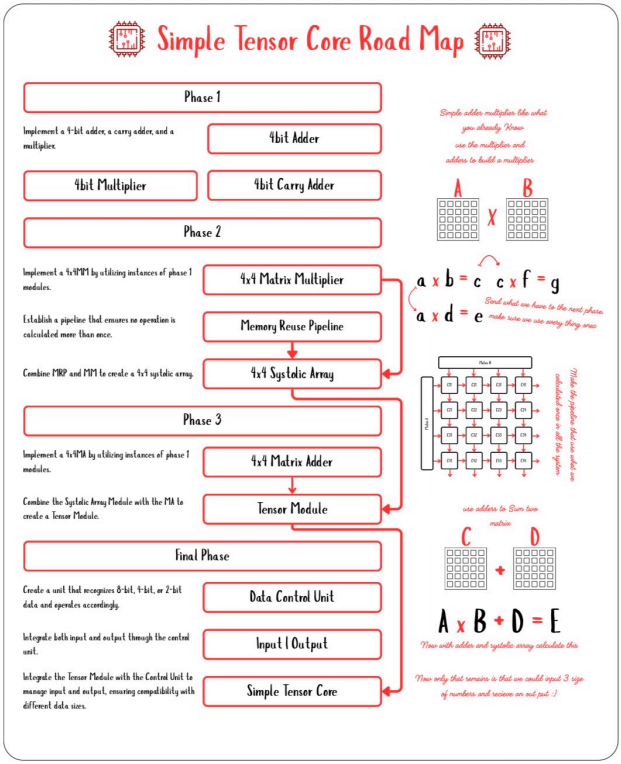

Figure 3: The Simple Tensor Core Architecture Roadmap.