Pipelined GPU

Shaileshram S

July 2025

1 Introduction

This document presents a comprehensive overview of my design and analysis of a pipelined GPU, based on the `tinyGPU` repository. The high-level data flow architecture is described as follows:

In my implementation, I employed a dual-core design. The top-level module comprises:

1. Data Memory Controller
2. Program Memory
3. Data Memory (simulated in the testbench)
4. Two processing cores
5. Device Control Register (DCR)
6. Dispatcher

To enable pipelining, I made extensive modifications across nearly all modules of the repository. The original design relied heavily on finite state machines (FSMs); I restructured it to adopt a pipeline-based approach with dedicated pipeline registers. However, pipelining this type of GPU is non-trivial due to bottlenecks in accessing both program and data memory. In my design, pipeline progression is gated by the slower of the two accesses—data memory or program memory.

To eliminate program memory contention, I introduced two independent program memory channels, one for each core, thereby removing the need for a program memory controller. Consequently, the critical path is now determined solely by the data memory controller.

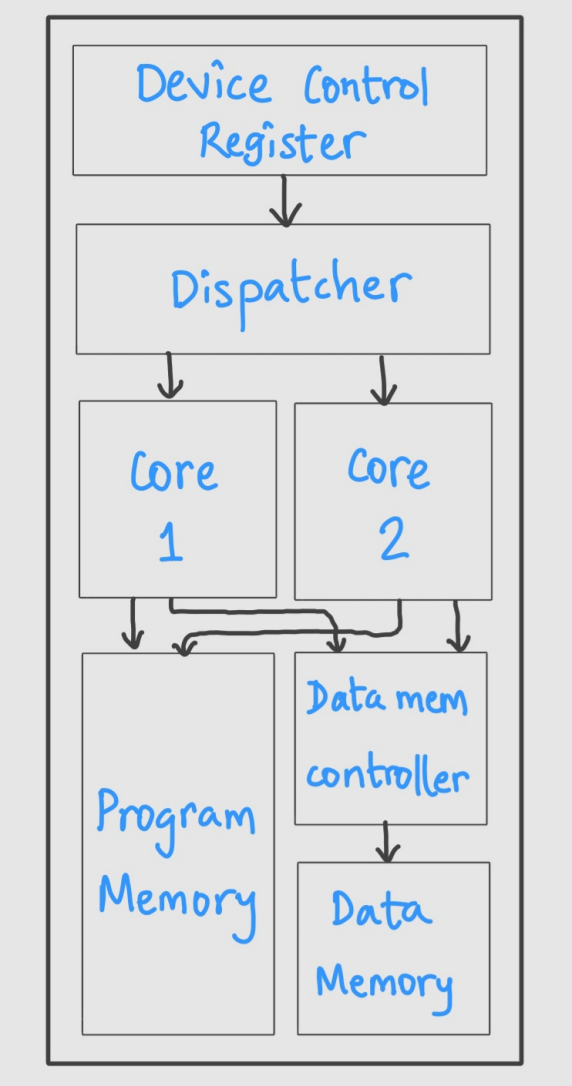

Figure 1: Top-Level Diagram

The pipeline is organized into four distinct stages:

1. Instruction Fetch

2. Instruction Decode

3. Instruction Execute / Memory Access

4. Register Writeback

I selected this 4-stage pipeline based on the instruction set architecture (ISA). Specifically, memory instructions in this design do not require ALU operations to compute addresses. This allowed for a more efficient implementation with a single data forwarding unit and a single hazard detection unit, reducing complexity.

The architecture of an individual core is shown below:

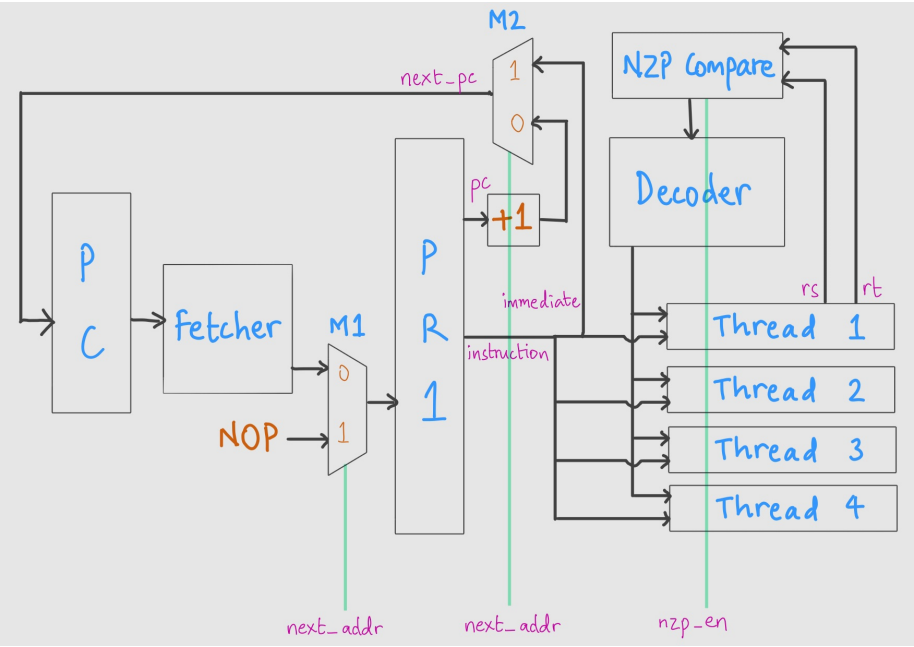

Figure 2: Core Architecture

Figure 2 depicts the logic unique to each core. The components labeled as NZP Compare, multiplexer M2, and the PC+1 block are collectively implemented as a unified "Next Address" block. This block determines the next value of the program counter (PC) based on relevant input signals.

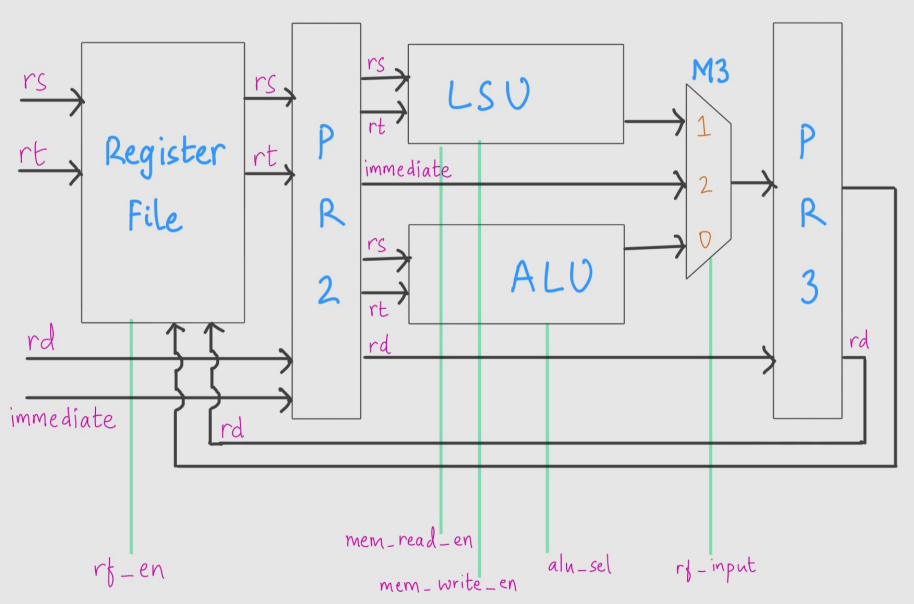

Figure 3: Thread-Specific Logic

Figure 3 illustrates the core logic that is specific to each thread within a core.

I implemented a load-store unit (LSU) state checker that acts as the driving mechanism for pipeline progression. Rather than advancing the pipeline solely on the rising edge of the clock, each pipeline register evaluates whether any LSU is currently waiting. This prevents premature progression and ensures correctness in memory operations.

To implement data forwarding and avoid unnecessary pipeline stalls, the following modified datapath is used:

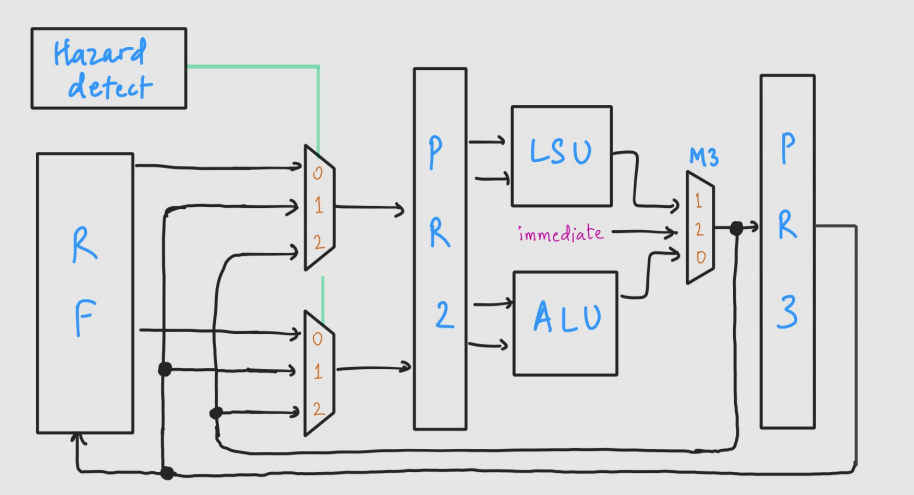

Figure 4: Core with Data Forwarding and Hazard Detection

A snapshot of the simulation waveform, demonstrating the execution of several ALU instructions, is shown below:

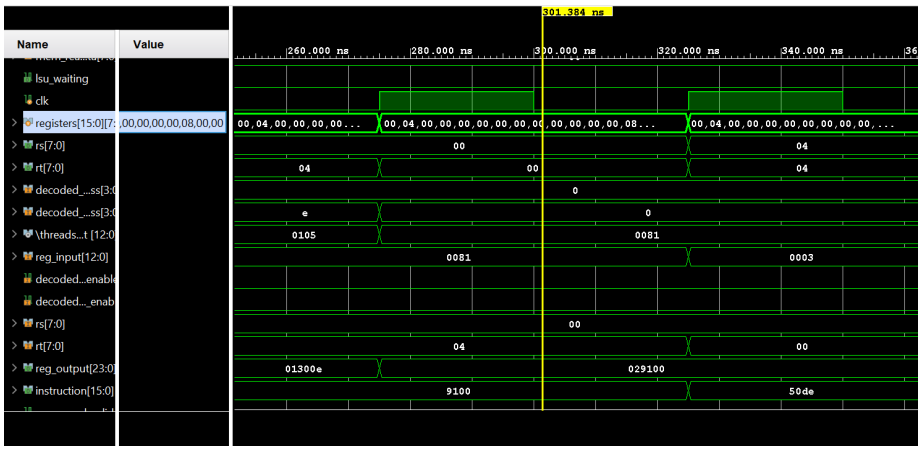

Figure 5: ALU Instruction Testbench

The corresponding test program is as follows:

```
16'b1001001000001000;    CONST R2, #8
16'b0011000000001110;    ADD R0, R0, %threadIdx
16'b1001000100000000;    CONST R1, #0
16'b0101000011011110;    MUL R0, blockIdx, blockDim
```

`16'b1001001100010000;` CONST R3, #16
`16'b1111000000000000;` RET

This simple program is designed to validate that the pipeline executes correctly across threads. It performs the following steps: assigns 8 to R2, updates R0 by adding the thread index, sets R1 to 0, calculates and stores the product of block ID and block dimension in R0, assigns 16 to R3, and terminates with a `RET` instruction.

To handle program termination, I introduced additional logic that freezes the PC upon execution of a `RET` instruction. The PC register is governed by a control signal `decoded_ret`, which, when asserted high, prevents further updates to the PC—effectively halting the thread until restarted.

The table below lists the values assigned to various control signals (highlighted in green in the diagrams) for each instruction type during the decode stage:

Instruction	next_addr	rf_input	alu_sel	rf_en	nzp_en	mem_read_en	mem_write_en	decoded_rt
NOP	0	00	00	0	0	0	0	0
BRnzp	1	00	00	0	0	0	0	0
CMP	0	00	00	0	1	0	0	0
ADD	0	00	00	1	0	0	0	0
SUB	0	00	01	1	0	0	0	0
MUL	0	00	10	1	0	0	0	0
DIV	0	00	11	1	0	0	0	0
LDR	0	01	00	1	0	1	0	0
STR	0	00	00	0	0	0	1	0
CONST	0	10	00	1	0	0	0	0
RET	0	00	00	0	0	0	0	1