TROY UNIVERSITY

CS 3365 - FALL 2024

Introduction to Computer Organization and Architecture

Group Assignment

Implementation of a minimalist GPU design using Verilog

Student:

Bui Dang Quoc An
Pham Tien Dat
Le Quang Huy
Nguyen Duy Khoi
Ngo Thanh Trung

Lecturer:

Dr. Nguyen Dinh Han

December 15, 2024

Abstract

This project focuses on building a simple GPU using Verilog to help understand how GPUs work and process tasks in parallel. The design includes key components like a control unit, scheduler, fetcher, decoder, compute cores,... Each compute core handles tasks with basic modules like an arithmetic-logic unit (ALU), program counter, and register files, allowing multiple threads to run instructions at the same time. The project uses a simple instruction set for operations like math, memory access, and control flow, making it easy to test and learn. This streamlined approach provides a clear starting point for exploring GPU architecture and parallel computing concepts.

Table of Contents

Program Documentation

1 Introduction

Graphics Processing Units (GPUs) are specialized hardware components designed to handle computationally intensive tasks, particularly those involving parallel processing. Initially developed for rendering graphics in video games, GPUs have evolved into versatile processors capable of accelerating a wide range of applications, including machine learning, scientific simulations, and big data analytics. Their ability to process thousands of threads simultaneously makes them indispensable in modern computing.

Unlike Central Processing Units (CPUs), which are optimized for sequential task execution, GPUs excel at parallelism. This is achieved through their architecture, which consists of numerous smaller cores that can execute tasks concurrently. This parallelism enables GPUs to handle large datasets and perform complex calculations more efficiently than CPUs in certain scenarios.

In this report, we aim to explore the architecture of GPUs and simulate their core components using Verilog, a hardware description language. By simulating key modules such as the Arithmetic Logic Unit (ALU), Decoder, Fetcher, Scheduler, and Compute Core, we gain a deeper understanding of how GPUs function and how their architecture supports high-performance parallel computing.

2 GPU Architecture

The architecture of a GPU is designed to maximize parallelism and throughput. It consists of several interconnected components that work together to execute tasks efficiently.

2.1 Core Compute

The Core Compute section is the heart of the GPU, where the actual computation takes place. It is responsible for executing instructions and performing arithmetic and logical operations. The main components of the Core Compute are:

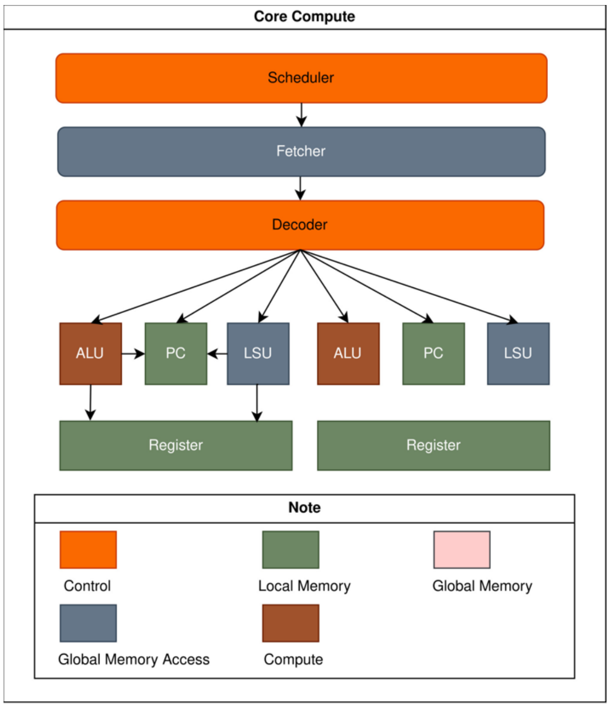


Figure 1: A Diagram of Compute Core

- **Scheduler:** The Scheduler is responsible for managing and distributing threads to the compute units. It ensures that all available resources are utilized efficiently by assigning tasks to idle threads and balancing the workload across the GPU cores.
- **Fetcher:** The Fetcher retrieves instructions from the instruction memory based on the Program Counter (PC). It ensures that the correct instructions are fetched and passed to the Decoder for further processing.
- **Decoder:** The Decoder interprets the fetched instructions and extracts the necessary fields, such as the opcode, source registers, destination registers, and immediate values. This information is then passed to the appropriate compute units for execution.
- **ALU (Arithmetic Logic Unit):** The ALU performs arithmetic and logical operations, such as addition, subtraction, multiplication, and comparisons. Each compute unit in the GPU contains multiple ALUs to handle parallel computations.
- **PC (Program Counter):** The Program Counter keeps track of the address of the current instruction being executed. It is updated after each instruction fetch to point to the next instruction.
- **LSU (Load/Store Unit):** The Load/Store Unit manages memory operations, such as loading data from memory into registers or storing data from registers back into memory. It handles both local memory (specific to a thread) and global memory (shared across threads).
- **Register:** Registers are small, fast storage units used to temporarily hold data during computations. Each thread has its own set of registers to store intermediate results and operands.

2.2 GPU Core

The GPU Core is a collection of multiple Compute Cores, each capable of executing tasks independently or in coordination with other cores.

- **Device Control Register:** This component manages the state and configuration of the GPU. It allows the host system to control the GPU's operation and monitor its status.
- **Dispatcher:** The Dispatcher assigns tasks to the Compute Cores. It ensures that workloads are distributed evenly across the cores to maximize performance and minimize idle time.
- **Cache:** The Cache is a small, high-speed memory that stores frequently accessed data and instructions. It reduces the latency of memory operations by providing faster access to data compared to global memory.
- **Program Memory Controller and Data Memory Controller:** These controllers manage access to the program memory and data memory, respectively. They ensure that instructions and data are fetched and stored efficiently, minimizing bottlenecks in memory access.

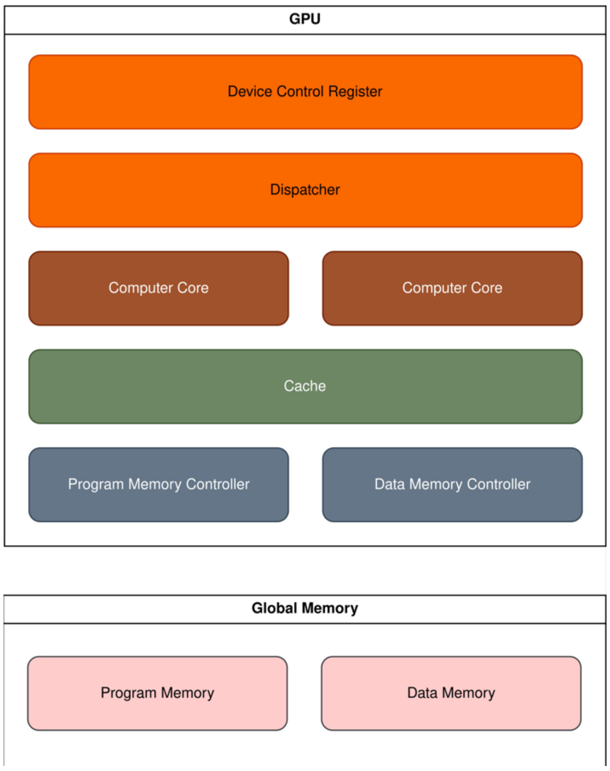


Figure 2: A GPU with common components

2.3 Global Memory

The Global Memory is a shared memory space accessible by all Compute Cores. It is used to store program instructions and data required for computations.

- **Program Memory:** This memory stores the instructions that the GPU executes. It is read by the Fetcher to retrieve the next instruction for execution.
- **Data Memory:** This memory stores the input data for computations and the results produced by the GPU. It is accessed by the Load/Store Unit during memory operations.

2.4 Key Features of GPU Architecture

The GPU architecture is characterized by several key features that enable its high performance:

1. **Massive Parallelism**: GPUs consist of thousands of small cores that can execute threads simultaneously.
2. **Thread Management**: The Scheduler and Dispatcher work together to manage threads efficiently.
3. **Memory Hierarchy**: GPUs use a hierarchical memory system to optimize memory access and reduce latency.
4. **Scalability**: The modular design of GPUs allows them to scale easily by adding more Compute Cores.
5. **Energy Efficiency**: GPUs are designed to perform many computations per watt of power consumed.

3 ALU Simulation in GPU

The ALU (Arithmetic Logic Unit) is a crucial component in any processor, including the GPU. The ALU performs arithmetic and logical operations on input operands, determining the outcome of calculations. The ALU simulation in this project includes basic operations such as addition, subtraction, multiplication, and comparison, helping us understand how the ALU interacts with operands during the GPU's computation process.

3.1 ALU Simulation Description

To simulate the ALU's operation, we have built a testbench with the Verilog code below. This testbench checks basic operations like ADD, SUB, MUL, and CMP.

1. **opcode**: The instruction specifying the operation to be performed (ADD, SUB, MUL, CMP).
2. **operand_a and operand_b**: The two input operands for the operation.
3. **immediate**: A constant that may be used in certain operations.
4. **result**: The result of the ALU operation.
5. **cmp_flag**: The comparison flag indicating the result of the CMP operation.

```verilog
`include "alu.v"  // Include the ALU module from alu.v

module alu_tb;

// Inputs
reg [3:0] opcode;
reg [`DATA_WIDTH-1:0] operand_a;
reg [`DATA_WIDTH-1:0] operand_b;
reg [7:0] immediate;

// Outputs
wire [`DATA_WIDTH-1:0] result;
wire cmp_flag;

// Instantiate the ALU
alu uut (
.opcode(opcode),
.operand_a(operand_a),
.operand_b(operand_b),
.immediate(immediate),
.result(result),
.cmp_flag(cmp_flag)
);

// Testbench stimulus
initial begin
// Initialize inputs
opcode = 4'b0000;
operand_a = 32'h00000000;
operand_b = 32'h00000000;
immediate = 8'h00;

// Display the results
$monitor("Time = %0t, opcode = %b, operand_a = %h,
   operand_b = %h, immediate = %h, result = %h, cmp_flag
   = %b",
$time, opcode, operand_a, operand_b, immediate, result,
   cmp_flag);

// Test operations
#10 opcode = `OP_ADD; operand_a = 32'h00000005;
   operand_b = 32'h00000003; immediate = 8'h02;
#10 opcode = `OP_SUB; operand_a = 32'h00000005;
   operand_b = 32'h00000003; immediate = 8'h02;
#10 opcode = `OP_MUL; operand_a = 32'h00000005;
   operand_b = 32'h00000003; immediate = 8'h00;
#10 opcode = `OP_CMP; operand_a = 32'h00000003;
   operand_b = 32'h00000005; immediate = 8'h00;
#10 opcode = 4'b1111; operand_a = 32'h00000000;
   operand_b = 32'h00000000; immediate = 8'h00;
```

```
        #10 $finish;
        end

        endmodule
```

3.2 Code Explanation

In the code above, we perform various operations and observe the results from the ALU. Specifically:

1. **Test case 1 (ADD)**: Adds the values of `operand_a` and `operand_b`.
2. **Test case 2 (SUB)**: Subtracts `operand_b` from `operand_a`.
3. **Test case 3 (MUL)**: Multiplies the two operands.
4. **Test case 4 (CMP)**: Compares the two operands to set the `cmp_flag`.
5. **Test case 5 (Default)**: Tests the behavior when an invalid opcode is used.

4 GPU Compute Core Simulation

In the GPU architecture, aside from the ALU, another critical component is the compute core, which handles complex calculations such as parallel processing, managing instructions from threads, and managing registers and memory. To simulate the operation of a compute core, we have built a simple Verilog testbench to examine control signals such as clk, reset, and halt, while also observing the execution of instruction programs.

4.1 Concept of the Compute Core

The compute core in a GPU processes program instructions, particularly those distributed across multiple threads. Each thread may have its own memory and perform operations in parallel, which significantly enhances computational performance. To manage threads, the compute core needs to be able to schedule, execute instructions, and manage registers for each thread.

4.2 Verilog Code: Compute Core Simulation

The following code simulates the basic operations of a compute core using a testbench. This testbench will examine clock cycles, reset signals, and control signals such as halt.

```
        // compute_core_tb.v

        module compute_core_tb;

        // Parameters
        parameter DATA_WIDTH = 32;      // Example for data width
        parameter NUM_THREADS = 4;      // Number of threads
        parameter REG_COUNT = 16;       // Number of registers
           per thread
        parameter ADDR_WIDTH = 4;       // Memory address width

        // Inputs
        reg clk;
        reg reset;

        // Outputs
        wire halt;

        // Instantiate the compute_core module
        compute_core uut (
        .clk(clk),
        .reset(reset),
        .halt(halt)
        );

        // Clock generation
        always begin
        #5 clk = ~clk;  // Generate clock with a period of 10
           time units
        end

        // Testbench stimulus
        initial begin
        // Initialize signals
        clk = 0;
        reset = 1;  // Start with reset

        // Display header
        $display("Time\tReset\tHalt\tPC[0]\tPC[1]\tPC[2]\tPC[3]");

        // Apply reset and release it after some time
        #10 reset = 0; // Release reset at time t = 0

        // Wait for some clock cycles and observe behavior
        #50; // Wait for a period

        // Test - simulate scheduling and instruction execution
        $display("Running computation...");
        #100; // Wait a bit to observe the operation

        // Apply reset again to see the reset behavior
        reset = 1;
        #10 reset = 0;
        #50;

        // End simulation
        $finish;
        end

        // Monitor the PC and halt signal for debugging
        always @(posedge clk) begin
        $display("%0t\t%0b\t%0b\t%h\t%h\t%h\t%h", $time, reset,
           halt,
        uut.pc[0], uut.pc[1], uut.pc[2], uut.pc[3]);
        end

        endmodule
```

4.3 Explanation of the Code

In the code above:

- **Clock generation:** We generate a clock signal (clk) with a period of 10 time units, simulating the continuous operation of the system.
- **Reset:** Initially, we apply a reset to the system and then release it after a certain period.
- **Behavior checking:** After releasing the reset, we simulate computation operations and observe signals such as halt, as well as the state of instruction programs through the values in the register pc[0], pc[1], pc[2], and pc[3].
- **Monitoring:** Every clock cycle, we print the status information to track the changes in the signals.

4.4 Results and Observations

The testbench will help verify whether the control signals operate correctly during the computation process, including system reset and instruction execution. You can observe the state of the instruction programs (PC) and monitor the halt signal to ensure that the compute core is functioning properly.

5 GPU Decoder Simulation

During the execution of instructions in a GPU, the first step is instruction decoding. Each instruction is encoded into bits, and the decoder is responsible for separating different parts of the instruction, such as the opcode, destination register, source register, and immediate values. This decoding process allows the GPU to determine what action to take with the corresponding data.

5.1 Concept of the Decoder

The decoder in a GPU works like an instruction splitter. It takes in a bit string of an instruction and separates it into various fields such as:

- **Opcode:** The operation code, which determines the type of operation to perform.

- **Dest_Reg:** The destination register, where the result of the operation will be stored.
- **Src_Reg:** The source register, which provides the input data for the operation.
- **Immediate:** The immediate value, which is a constant that can be used in the operation.

5.2 Verilog Code: Decoder Simulation

Below is the testbench code that simulates a simple decoder. This testbench uses a 16-bit instruction value and checks the output signals after decoding.

```verilog
// decoder_tb.v

module decoder_tb;

// Parameters
parameter DATA_WIDTH = 16;  // Instruction width (16
   bits)

// Inputs
reg [DATA_WIDTH-1:0] instruction;

// Outputs
wire [3:0] opcode;
wire [3:0] dest_reg;
wire [3:0] src_reg;
wire [7:0] immediate;

// Instantiate the decoder module
decoder uut (
.instruction(instruction),
.opcode(opcode),
.dest_reg(dest_reg),
.src_reg(src_reg),
.immediate(immediate)
);

// Testbench stimulus
initial begin
// Initialize signals
instruction = 16'b0001_1010_1100_0011; // Sample
   instruction value

// Display header
$display("Time\tInstruction\tOpcode\tDest_Reg\tSrc_Reg\tImmediate");

// Apply different instruction values and monitor output
#10 instruction = 16'b1001_0101_0110_0111;  // Change
   instruction value
```

```
        #10 instruction = 16'b0110_1111_0001_0010;  // Another
           instruction value
        #10 instruction = 16'b1110_0000_1000_1111;  // Yet
           another value

        // End simulation
        #10 $finish;
        end

        // Monitor outputs during simulation
        always @(instruction) begin
        $display("%0t\t%h\t\t%h\t\t%h\t\t%h\t\t%h",
        $time, instruction, opcode, dest_reg, src_reg,
           immediate);
        end

        endmodule
```

5.3 Explanation of the Code

In the code above:

- **Inputs:** The input instruction is 16 bits wide. This is the instruction that the decoder will break into different fields.
- **Outputs:** The decoder provides the following output values after decoding:
  - **Opcode (4 bits):** The operation code.
  - **Dest_Reg (4 bits):** The destination register.
  - **Src_Reg (4 bits):** The source register.
  - **Immediate (8 bits):** The immediate value.
- **Testbench stimulus:** Various sample instructions are provided at each clock cycle to test the correctness of the decoding process. Every 10 time units, the instruction value is updated to test different scenarios.
- **Monitor:** Each time the instruction value changes, the opcode, dest_reg, src_reg, and immediate signals are displayed to monitor the decoding process.

5.4 Results and Observations

The testbench allows you to observe the changes in the output signals whenever the input instruction changes. This helps verify whether the decoder correctly decodes the instruction into its components. You can easily track the decoding process and check each field in the instruction through the printed values.

6 Fetcher Simulation in GPU

In a GPU system, one of the critical steps in the program execution process is fetching instructions from memory. Each instruction from memory is fetched and prepared for decoding and execution. The fetcher module is responsible for retrieving instructions from memory based on the Program Counter (PC) address. This address determines the location of the instruction in memory, and the fetcher module returns the corresponding instruction for processing.

6.1 Concept of Fetcher

The fetcher is a crucial module in the architecture of processors, including GPUs. Its function is to read instructions from the instruction memory based on the value of the Program Counter (PC). As the PC changes, the fetcher fetches the new instruction from memory based on the given address.

6.2 Verilog Code: Fetcher Simulation

The following is the Verilog code for the fetcher simulation.

```
        module fetcher_tb;

        // Parameters
        parameter ADDR_WIDTH = 16;      // Address width (16
           bits)
        parameter DATA_WIDTH = 16;      // Data width (16 bits)

        // Inputs
        reg [ADDR_WIDTH-1:0] pc_in;     // Program counter
           address
        reg [DATA_WIDTH-1:0] instr_mem [0:15];  // Instruction
           memory

        // Outputs
        wire [DATA_WIDTH-1:0] instruction;  // Instruction
           fetched from memory

        // Instantiate the fetcher module
        fetcher uut (
        .pc_in(pc_in),
        .instruction(instruction),
        .instr_mem(instr_mem)
        );

        // Testbench stimulus
        initial begin
        // Initialize instruction memory (example with 16
           instructions, each 16 bits)
        instr_mem[0] = 16'hA5A5;  // Instruction 0
        instr_mem[1] = 16'h5A5A;  // Instruction 1
        instr_mem[2] = 16'h1234;  // Instruction 2
        instr_mem[3]  = 16'hDEAD;  // Instruction 3
        instr_mem[4]  = 16'h8765;  // Instruction 4
        instr_mem[5]  = 16'hABCD;  // Instruction 5
        instr_mem[6]  = 16'h1122;  // Instruction 6
        instr_mem[7]  = 16'hAABB;  // Instruction 7
        instr_mem[8]  = 16'hDEAD;  // Instruction 8
        instr_mem[9]  = 16'hF00D;  // Instruction 9
        instr_mem[10] = 16'hC0FF; // Instruction 10
        instr_mem[11] = 16'h0000; // Instruction 11
        instr_mem[12] = 16'hCAFE; // Instruction 12
        instr_mem[13] = 16'h1234; // Instruction 13
        instr_mem[14] = 16'h4321; // Instruction 14
        instr_mem[15] = 16'h9999; // Instruction 15

        // Display header
        $display("Time\tPC\tInstruction");

        // Apply different values of pc_in and observe the
           instruction fetched
        #10 pc_in = 16'b0000000000000000; // Fetch instruction
           at address 0
        #10 pc_in = 16'b0000000000000001; // Fetch instruction
           at address 1
        #10 pc_in = 16'b0000000000000010; // Fetch instruction
           at address 2
        #10 pc_in = 16'b0000000000000011; // Fetch instruction
           at address 3
        #10 pc_in = 16'b0000000000000100; // Fetch instruction
           at address 4
        #10 pc_in = 16'b0000000000000101; // Fetch instruction
           at address 5
        #10 pc_in = 16'b0000000000000110; // Fetch instruction
           at address 6
        #10 pc_in = 16'b0000000000000111; // Fetch instruction
           at address 7
        #10 pc_in = 16'b0000000000001000; // Fetch instruction
           at address 8
        #10 pc_in = 16'b0000000000001001; // Fetch instruction
           at address 9
        #10 pc_in = 16'b0000000000001010; // Fetch instruction
           at address 10
        #10 pc_in = 16'b0000000000001011; // Fetch instruction
           at address 11
        #10 pc_in = 16'b0000000000001100; // Fetch instruction
           at address 12
        #10 pc_in = 16'b0000000000001101; // Fetch instruction
           at address 13
        #10 pc_in = 16'b0000000000001110; // Fetch instruction
           at address 14
        #10 pc_in = 16'b0000000000001111; // Fetch instruction
           at address 15
```

```verilog

        // End simulation
        #10 $finish;
        end

        // Monitor the fetched instruction
        always @(pc_in) begin
        $display("%0t\t%h\t%h", $time, pc_in, instruction);
        end

        endmodule
```

6.3 Explanation of the Code

- **Inputs:**
  - `pc_in`: This is the Program Counter (PC) input, which determines the address in memory from which the instruction will be fetched.
  - `instr_mem`: This is the memory that stores the instructions. It is initialized with 16 sample instruction values, each 16 bits long.
- **Outputs:**
  - `instruction`: The instruction fetched from memory based on the PC address.
- **Testbench stimulus:** The instruction memory `instr_mem` is initialized with sample values. Then, the value of `pc_in` changes over multiple cycles to test the instruction fetching from different addresses in memory. Every 10 time units, the `pc_in` value is updated to fetch an instruction from a different address.
- **Monitor:** Whenever the `pc_in` value changes, the corresponding instruction is displayed, allowing us to track the instruction fetch from memory.

6.4 Results and Observations

This testbench helps verify the accuracy of the instruction-fetching process from the instruction memory. Every time the PC value changes, a new instruction is fetched from memory, ensuring that the fetcher module is functioning correctly and fetching instructions from the right memory address.

7 Scheduler Simulation in a Multithreading System

In a multithreading system, the scheduler module is responsible for allocating threads to the CPU or other resources. The scheduler takes input signals indicating which threads are active and selects one thread to execute during each clock cycle. The testbench below simulates the input signals and observes the output of the scheduler module, which includes scheduling the active threads.

7.1 Concept of Scheduler

The scheduler is the module responsible for scheduling threads within the system. The signal `active_threads` indicates which threads are active (1 means active, 0 means inactive). Based on this signal, the scheduler selects a thread to execute in each clock cycle. The output `scheduled_thread` is the ID of the thread that has been scheduled for execution.

7.2 Verilog Code: Scheduler Simulation

Below is the Verilog code for the scheduler simulation.

```
        module scheduler_tb;

        // Parameters
        parameter NUM_THREADS = 4;  // Assuming there are 4
           threads
        parameter THREAD_ID_WIDTH = 3;  // Set the thread ID
           width to 3 bits

        // Inputs
        reg clk;
        reg reset;
        reg [NUM_THREADS-1:0] active_threads;  // Signal
           indicating which threads are active

        // Outputs
        wire [THREAD_ID_WIDTH-1:0] scheduled_thread;  // The
           scheduled thread

        // Instantiate the scheduler module
        scheduler #(
        .NUM_THREADS(NUM_THREADS)
        ) uut (
        .clk(clk),
        .reset(reset),
        .active_threads(active_threads),
        .scheduled_thread(scheduled_thread)
        );

        // Clock generation
        always begin
        #5 clk = ~clk;  // Clock cycle with 10 time units
        end

        // Testbench stimulus
        initial begin
        // Initialize signals
        clk = 0;
        reset = 1;
        active_threads = 4'b1111;  // All threads are active

        // Display header
        $display("Time\tReset\tActive Threads\tScheduled
           Thread");

        // Apply reset and release it after a while
        #10 reset = 0;  // Release reset at time t = 10

        // Run tests with different active_threads values
        #10 active_threads = 4'b1110;  // Thread 3 is inactive
        #10 active_threads = 4'b1101;  // Thread 2 is inactive
        #10 active_threads = 4'b1011;  // Thread 1 is inactive
        #10 active_threads = 4'b1000;  // Only thread 0 is active

        // Run with all threads active
        #10 active_threads = 4'b1111;

        // Test with reset again
        #10 reset = 1;  // Apply reset
        #10 reset = 0;  // Release reset

        // End simulation after enough tests
        #50 $finish;
        end

        // Monitor output
        always @(posedge clk) begin
        $display("%0t\t%0b\t%0b\t%0d", $time, reset,
           active_threads, scheduled_thread);
        end

        endmodule
```

7.3 Explanation of the Code

- **Inputs:**
  - `clk`: The clock signal controlling the operation of the simulation.
  - `reset`: The reset signal to initialize the scheduler's state.
  - `active_threads`: A 4-bit signal indicating which threads are active (1 means active, 0 means inactive).
- **Outputs:**
  - `scheduled_thread`: The ID of the thread chosen by the scheduler to be executed.
- **Clock Generation:**
  - The clock cycle is generated with a period of 10 time units (5 time units for each clock state).

- **Testbench Stimulus:**
  - `reset` is asserted during the first 10 time units and then deasserted.
  - `active_threads` is changed to simulate different scenarios:
    - All threads active.
    - Some threads inactive.
    - Only one thread active.
  - After modifying the signals, the reset signal is applied again to test the system in the reset state.
- **Monitor Output:**
  - Whenever a positive edge of the clock (`posedge clk`) occurs, the program will display the simulation time, the value of the reset, `active_threads`, and `scheduled_thread` signals.

7.4 Results and Observations

The scheduler will select a thread to execute based on which threads are active during each clock cycle. When one or more threads are inactive (with a 0 bit in `active_threads`), the scheduler will only schedule the active threads.

Changing the state of `active_threads` allows observation of how the scheduler responds and selects the correct thread.

7.5 Conclusion

Those codes helps simulate and verify the behavior of the scheduler in a multithreading system. By modifying the `active_threads` states and triggering the reset signal, you can check the correctness of the thread scheduling within the system.

Test Bench Documentation

8 Testbench

A testbench is a simulated environment used to verify and test the functionality of hardware designs, particularly in digital circuit design using languages like Verilog or VHDL. It is not part of the actual hardware but is used during the design and verification process to simulate how the design would behave in a real-world scenario.

In essence, a testbench:

1.Stimulates Inputs: It provides input signals to the design (usually through a set of test vectors) to simulate different conditions and scenarios the hardware may encounter.

2.Monitors Outputs: It observes and checks the outputs generated by the design in response to the inputs.

3.Checks for Correct Behavior: It compares the actual outputs against the expected behavior (the specification or golden model) to detect bugs or issues.

4.Helps in Debugging: It allows designers to debug and troubleshoot their designs without needing physical hardware, speeding up the development process.

A testbench typically consists of:

1.Clock Generation: A clock signal for synchronous designs.

2.Reset Logic: To initialize the design at the start of the simulation.

3.Stimuli Generation: Input signals are provided based on the test cases.

4.Monitoring/Assertion: Checking outputs, often with assert statements to automatically verify if the design behaves as expected. article amsmath

graphicx

9 ALU

9.1 Key Features of the ALU Module

1.Opcode Control:

The ALU's behavior is controlled by the opcode input, which determines the operation (ADD, SUB, MUL, CMP, etc.).

2.Operand Handling:

The ALU accepts two operands (operand_a and operand_b), which can be used for arithmetic or comparison operations.

3.Immediate Input:

The immediate input provides additional data to the ALU, though it is not used in all operations (such as ADD, SUB, or MUL).

4.Output:

-The ALU outputs a result, which holds the result of the operation. -The cmp_flag indicates the result of comparison operations (e.g., for CMP, it could indicate whether operand_a is less than, equal to, or greater than operand_b).

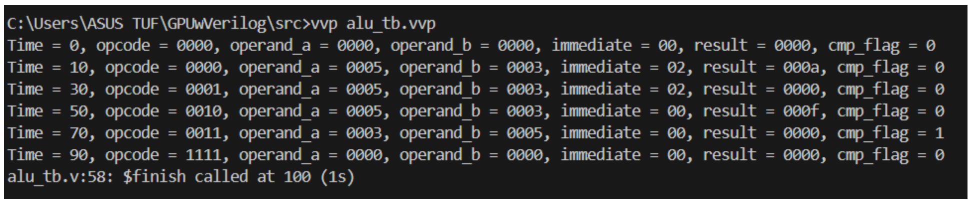
9.2 alu_tb.v

Testbench for ALU

ALU Testbench Code

```
 // `include "alu.v"  // Include the ALU module from alu.v

module alu_tb;

    // Inputs
    reg [3:0] opcode;
    reg [`DATA_WIDTH-1:0] operand_a;
    reg [`DATA_WIDTH-1:0] operand_b;
    reg [7:0] immediate;

    // Outputs
    wire [`DATA_WIDTH-1:0] result;
    wire cmp_flag;

    // Instantiate the ALU (Using the included module from alu.v)
    alu uut (
        .opcode(opcode),
        .operand_a(operand_a),
        .operand_b(operand_b),
        .immediate(immediate),
        .result(result),
        .cmp_flag(cmp_flag)
    );

    // Testbench stimulus
    initial begin
        // Initialize inputs
        opcode = 4'b0000;
        operand_a = 32'h00000000;
        operand_b = 32'h00000000;
        immediate = 8'h00;

        // Display the results
        $monitor("Time = %0t, opcode = %b, operand_a = %h,
           operand_b = %h, immediate = %h, result = %h, cmp_flag
           = %b",
                 $time, opcode, operand_a, operand_b, immediate,
                    result, cmp_flag);

        // Test case 1: ADD operation
        #10 opcode = `OP_ADD; operand_a = 32'h00000005;
           operand_b = 32'h00000003; immediate = 8'h02;
        #10; // Wait for 10 time units

        // Test case 2: SUB operation
        #10 opcode = `OP_SUB; operand_a = 32'h00000005;
           operand_b = 32'h00000003; immediate = 8'h02;
        #10; // Wait for 10 time units

        // Test case 3: MUL operation
        #10 opcode = `OP_MUL; operand_a = 32'h00000005;
           operand_b = 32'h00000003; immediate = 8'h00;
        #10; // Wait for 10 time units

        // Test case 4: CMP operation (comparison)
        #10 opcode = `OP_CMP; operand_a = 32'h00000003;
           operand_b = 32'h00000005; immediate = 8'h00;
        #10; // Wait for 10 time units

        // Test case 5: Default operation
        #10 opcode = 4'b1111; operand_a = 32'h00000000;
           operand_b = 32'h00000000; immediate = 8'h00;
        #10; // Wait for 10 time units

        // End the simulation
        $finish;
    end

endmodule
```

9.3 Result.v



```
C:\Users\ASUS TUF\GPUwVerilog\src>vvp alu_tb.vvp
Time = 0, opcode = 0000, operand_a = 0000, operand_b = 0000, immediate = 00, result = 0000, cmp_flag = 0
Time = 10, opcode = 0000, operand_a = 0005, operand_b = 0003, immediate = 02, result = 000a, cmp_flag = 0
Time = 30, opcode = 0001, operand_a = 0005, operand_b = 0003, immediate = 02, result = 0000, cmp_flag = 0
Time = 50, opcode = 0010, operand_a = 0005, operand_b = 0003, immediate = 00, result = 000f, cmp_flag = 0
Time = 70, opcode = 0011, operand_a = 0003, operand_b = 0005, immediate = 00, result = 0000, cmp_flag = 1
Time = 90, opcode = 1111, operand_a = 0000, operand_b = 0000, immediate = 00, result = 0000, cmp_flag = 0
alu_tb.v:58: $finish called at 100 (1s)
```

Figure 3: Testbench Verilog Output

10 Decoder

10.1 Key Features of the decoder Module

1.Opcode Decoding:

The behavior of the Decoder is controlled by the top 4 bits of the instruction (opcode), which determine the type of operation to be executed (e.g., LOAD, STORE, ADD, SUB, JMP, etc.).

2.Destination Register Identification (Dest_Reg):

The Decoder extracts the next 4 bits of the instruction to identify the destination register (dest_reg), where the result of the operation will be stored.

3.Source Register Identification (Src_Reg):

The following 4 bits of the instruction specify the source register (src_reg) that contains the required data for the operation.

4.Immediate Decoding:

The lowest 8 bits of the instruction are extracted as the immediate value, providing additional data for certain instructions (e.g., ADDI, LOADI).

5.Outputs:

-opcode: Specifies the operation type to be executed.

-dest_reg: Identifies the destination register.

-src_reg: Identifies the source register.

-immediate: Supplies the immediate value if required by the instruction.

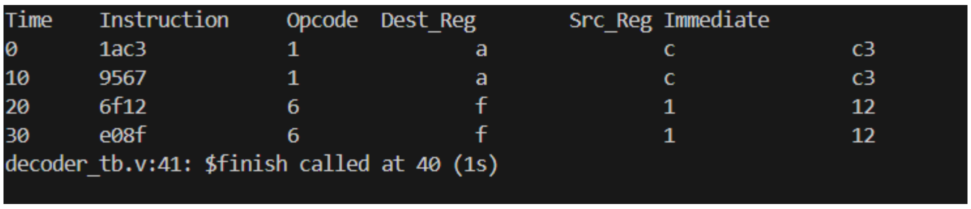
10.2 decoder_tb.v

Testbench for Decoder

Decoder Testbench Code

```
// decoder_tb.v


module decoder_tb;

    // Parameters
    parameter DATA_WIDTH = 16;  // Chi u  r ng  c a
       instruction (16 bit)

    // Inputs
    reg [DATA_WIDTH-1:0] instruction;

    // Outputs
    wire [3:0] opcode;
    wire [3:0] dest_reg;
    wire [3:0] src_reg;
    wire [7:0] immediate;

    // Instantiate the decoder module
    decoder uut (
        .instruction(instruction),
        .opcode(opcode),
        .dest_reg(dest_reg),
        .src_reg(src_reg),
        .immediate(immediate)
    );

    // Testbench stimulus
    initial begin
        // Initialize signals
        instruction = 16'b0001_1010_1100_0011; // M t gi t r instruction m u

        // Display header
        $display("Time\tInstruction\tOpcode\tDest_Reg\tSrc_Reg\tImmediate");

        // Apply different instruction values and monitor output
        #10 instruction = 16'b1001_0101_0110_0111;  // Thay i gi t r instruction
        #10 instruction = 16'b0110_1111_0001_0010;  // M t gi t r instruction kh c
        #10 instruction = 16'b1110_0000_1000_1111;  // Th m m t gi t r kh c

        // End simulation
        #10 $finish;
    end

    // Monitor outputs during simulation
    always @(instruction) begin
        $display("%0t\t%h\t\t%h\t\t%h\t\t%h\t\t%h",
                 $time, instruction, opcode, dest_reg, src_reg, immediate);
    end

endmodule
```

10.3 Result.v



```
Time      Instruction       Opcode  Dest_Reg         Src_Reg  Immediate
0         1ac3              1              a              c              c3
10        9567              1              a              c              c3
20        6f12              6              f              1              12
30        e08f              6              f              1              12
decoder_tb.v:41: $finish called at 40 (1s)
```

Figure 4: Testbench Verilog Output

11 Fetcher

11.1 Key Features of the fetcher Module

Accessing Memory via Program Counter (PC):

The behavior of the Fetcher module is controlled by the pc_in (Program Counter) input, which specifies the address of the instruction to be fetched from memory.

Instruction Memory (instr_mem):

-The Fetcher module uses instr_mem, a memory containing instructions (each 16 bits in size). -The instruction at the pc_in address is fetched and provided as output.

Instruction Output:

The instruction output holds the 16-bit instruction fetched from the instruction memory based on the pc_in address.

Sequential Address Operation:

The Fetcher fetches instructions from different addresses in the instruction memory as the pc_in value changes.

Simulation and Display:

Every change in pc_in triggers fetching a new instruction and displays the corresponding instruction using *display*.

11.2 fetcher_tb.v

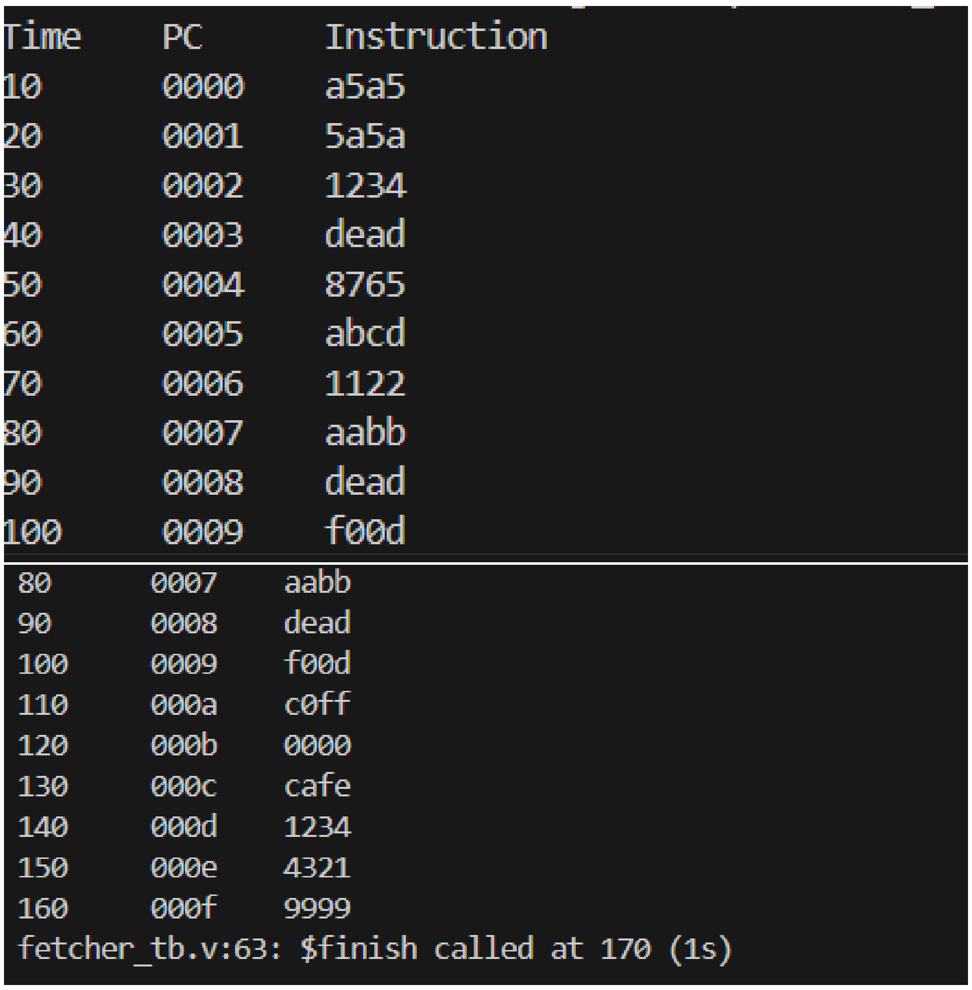
Testbench for Fetcher

Fetcher Testbench Code

```
module fetcher_tb;

   // Parameters
   parameter ADDR_WIDTH = 16;      // C h i u   r ng     a
       c h  (16 bit)
   parameter DATA_WIDTH = 16;      // C h i u   r ng   d
       l i u  (16 bit)

   // Inputs
   reg [ADDR_WIDTH-1:0] pc_in;    //    a     c h   c h   n g
       tr  n h
   reg [DATA_WIDTH-1:0] instr_mem [0:15];  //  B    n h    c h
       t h  (instruction memory)

   // Outputs
   wire [DATA_WIDTH-1:0] instruction;  //  L nh      c     fetch
       t    b    n h

   // Instantiate the fetcher module
   fetcher uut (
       .pc_in(pc_in),
       .instruction(instruction),
       .instr_mem(instr_mem)
   );

   // Testbench stimulus
   initial begin
```

```
        // Initialize instruction memory ( v   d    v i  16 c h  t h , m i  c h  t h  16 bit)
        instr_mem[0] = 16'hA5A5;  // Instruction 0
        instr_mem[1] = 16'h5A5A;  // Instruction 1
        instr_mem[2] = 16'h1234;  // Instruction 2
        instr_mem[3] = 16'hDEAD;  // Instruction 3
        instr_mem[4] = 16'h8765;  // Instruction 4
        instr_mem[5] = 16'hABCD;  // Instruction 5
        instr_mem[6] = 16'h1122;  // Instruction 6
        instr_mem[7] = 16'hAABB;  // Instruction 7
        instr_mem[8] = 16'hDEAD;  // Instruction 8
        instr_mem[9] = 16'hF00D;  // Instruction 9
        instr_mem[10] = 16'hC0FF; // Instruction 10
        instr_mem[11] = 16'h0000; // Instruction 11
        instr_mem[12] = 16'hCAFE; // Instruction 12
        instr_mem[13] = 16'h1234; // Instruction 13
        instr_mem[14] = 16'h4321; // Instruction 14
        instr_mem[15] = 16'h9999; // Instruction 15

        // Display header
        $display("Time\tPC\tInstruction");

        // Apply different values of pc_in and observe the instruction fetched
        #10 pc_in = 16'b0000000000000000; // Fetch instruction at address 0
        #10 pc_in = 16'b0000000000000001; // Fetch instruction at address 1
        #10 pc_in = 16'b0000000000000010; // Fetch instruction at address 2
        #10 pc_in = 16'b0000000000000011; // Fetch instruction at address 3
        #10 pc_in = 16'b0000000000000100; // Fetch instruction at address 4
        #10 pc_in = 16'b0000000000000101; // Fetch instruction at address 5
        #10 pc_in = 16'b0000000000000110; // Fetch instruction at address 6
        #10 pc_in = 16'b0000000000000111; // Fetch instruction at address 7
        #10 pc_in = 16'b0000000000001000; // Fetch instruction at address 8
        #10 pc_in = 16'b0000000000001001; // Fetch instruction at address 9
        #10 pc_in = 16'b0000000000001010; // Fetch instruction at address 10
        #10 pc_in = 16'b0000000000001011; // Fetch instruction at address 11
        #10 pc_in = 16'b0000000000001100; // Fetch instruction at address 12
        #10 pc_in = 16'b0000000000001101; // Fetch instruction
            at address 13
        #10 pc_in = 16'b0000000000001110; // Fetch instruction
            at address 14
        #10 pc_in = 16'b0000000000001111; // Fetch instruction
            at address 15

        // End simulation
        #10 $finish;
    end

    // Monitor the fetched instruction
    always @(pc_in) begin
        $display("%0t\t%h\t%h", $time, pc_in, instruction);
    end

endmodule
```

11.3 Result.v



```
Time      PC        Instruction
10        0000      a5a5
20        0001      5a5a
30        0002      1234
40        0003      dead
50        0004      8765
60        0005      abcd
70        0006      1122
80        0007      aabb
90        0008      dead
100       0009      f00d
```

```
80      0007    aabb
90      0008    dead
100     0009    f00d
110     000a    c0ff
120     000b    0000
130     000c    cafe
140     000d    1234
150     000e    4321
160     000f    9999
fetcher_tb.v:63: $finish called at 170 (1s)
```

Figure 5: Testbench Verilog Output

12 SCHEDULER

12.1 Key Features of the Scheduler Module

1.Controlled by Active Threads Signal:

-The behavior of the Scheduler is controlled by the active_threads input, which identifies the active threads (1 = active, 0 = inactive).

-The Scheduler selects an appropriate thread based on the status of the bits in this signal.

2.Reset Handling:

The reset signal allows the Scheduler to reinitialize. When reset is activated, the scheduled thread defaults back to thread 0.

3.Sequential Scheduling:

-The Scheduler selects threads in sequential order, starting with the lowest active bit in active_threads.

-If only one thread is active, the Scheduler will select that thread.

4.Output:

scheduled_thread: Outputs the ID of the scheduled thread, which ranges from 0 to NUM_THREADS-1.

5.Synchronized with Clock Signal:

The Scheduler operates synchronously with the clock signal clk, ensuring scheduling is executed accurately on each clock cycle.

12.2 scheduler_tb.v

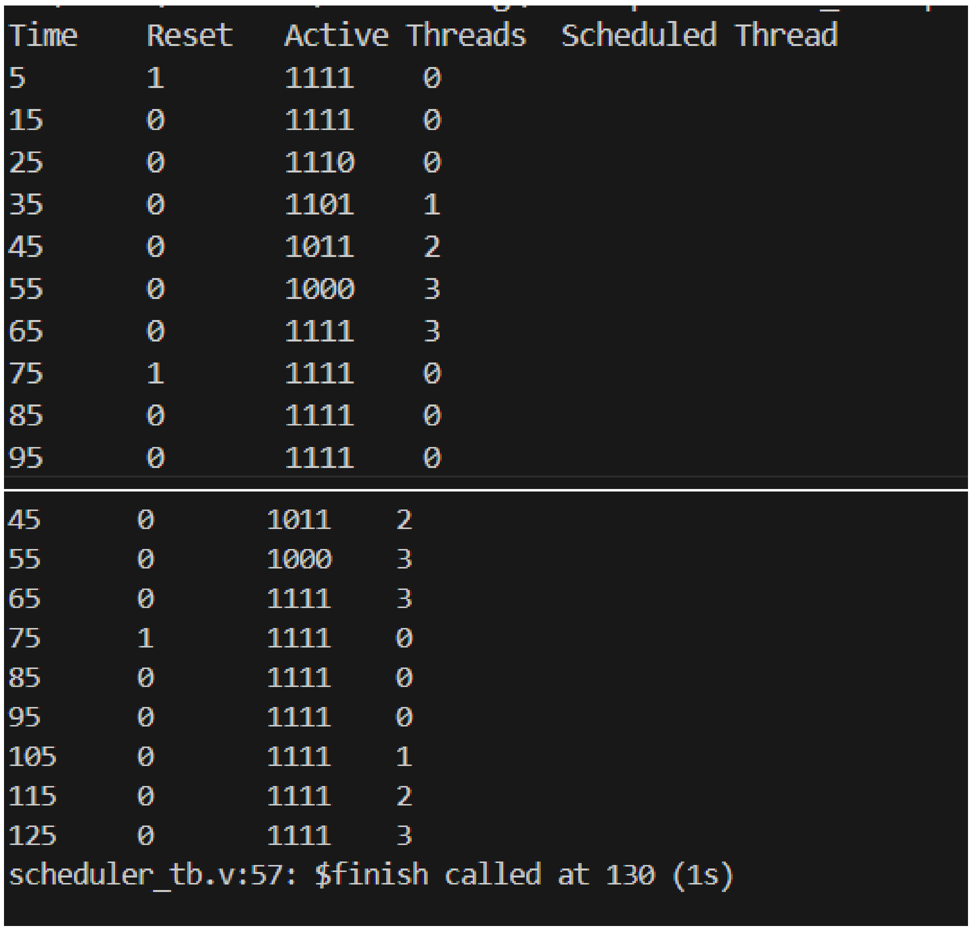
Testbench for Scheduler

Scheduler Testbench Code

```
module scheduler_tb;

    // Parameters
    parameter NUM_THREADS = 4;  // G i   s   c  4 threads
    parameter THREAD_ID_WIDTH = 3;  // S a   c h i u   r ng  ID thread th nh 3 bit

    // Inputs
    reg clk;
    reg reset;
    reg [NUM_THREADS-1:0] active_threads;  // T n  h i u  x c nh   c c thread c n  h o t   ng

    // Outputs
    wire [THREAD_ID_WIDTH-1:0] scheduled_thread;  // Thread  c    l p   l ch

    // Instantiate the scheduler module
    scheduler #(
        .NUM_THREADS(NUM_THREADS)
    ) uut (
        .clk(clk),
        .reset(reset),
        .active_threads(active_threads),
        .scheduled_thread(scheduled_thread)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;  // Chu  k      ng     h   10   n    v   t h i  gian
    end

    // Testbench stimulus
    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;
        active_threads = 4'b1111;  //  T t    c    c c thread   u    h o t    ng

        // Display header
        $display("Time\tReset\tActive Threads\tScheduled Thread");

        // Apply reset and release it after a while
        #10 reset = 0;  // Release reset  t i   t h i    im   t  = 10

        //  C h y  t h   v i  c c active_threads kh c nhau
        #10 active_threads = 4'b1110;  // Thread 3 kh ng  h o t  ng
        #10 active_threads = 4'b1101;  // Thread 2 kh ng  h o t  ng
        #10 active_threads = 4'b1011;  // Thread 1 kh ng  h o t  ng
        #10 active_threads = 4'b1000;  //  C h   c  thread 0  h o t    ng

        //  C h y  v i  t t  c   c c thread   u    h o t  ng
        #10 active_threads = 4'b1111;

        // Test with reset again
        #10 reset = 1;  //  T h c  h i n  reset
        #10 reset = 0;  //  T h   reset

        //  K t  th c  m   p h ng  sau khi test       c c  t nh  h u ng
        #50 $finish;
```

```verilog
    end

    // Monitor output
    always @(posedge clk) begin
        $display("%0t\t%0b\t%0b\t%0d", $time, reset,
            active_threads, scheduled_thread);
    end

endmodule
```

12.3 TestBench/Result.v



```
Time     Reset    Active Threads  Scheduled Thread
5        1        1111     0
15       0        1111     0
25       0        1110     0
35       0        1101     1
45       0        1011     2
55       0        1000     3
65       0        1111     3
75       1        1111     0
85       0        1111     0
95       0        1111     0
```

```
45       0        1011     2
55       0        1000     3
65       0        1111     3
75       1        1111     0
85       0        1111     0
95       0        1111     0
105      0        1111     1
115      0        1111     2
125      0        1111     3
scheduler_tb.v:57: $finish called at 130 (1s)
```

Figure 6: Testbench Verilog Output

13 COMPUTE CORE

13.1 Key Features of the Compute Core Module

1.Controlled by Active Threads Signal:
    1.Clock-Controlled Operation:

The behavior of the Compute Core is synchronized with the clock signal (clk), ensuring that all operations, including instruction execution and thread scheduling, progress accurately with each clock cycle.

2.Reset Handling:

The reset signal initializes the module, resetting all internal states, including Program Counters (PC) of all threads, and ensuring a clean starting point for computation.

3.Thread Management:

-The module manages multiple threads (e.g., 4 threads) simultaneously, each with its own Program Counter (PC).

-It tracks the state and progress of all threads during execution.

4.Halt Signal:

The halt signal indicates the completion of computation. When the module finishes executing instructions or meets specific conditions, the halt signal is asserted.

5.Program Counter Updates:

The module updates the Program Counter (PC) for each thread sequentially or based on scheduling logic, enabling efficient execution of instructions for multiple threads.

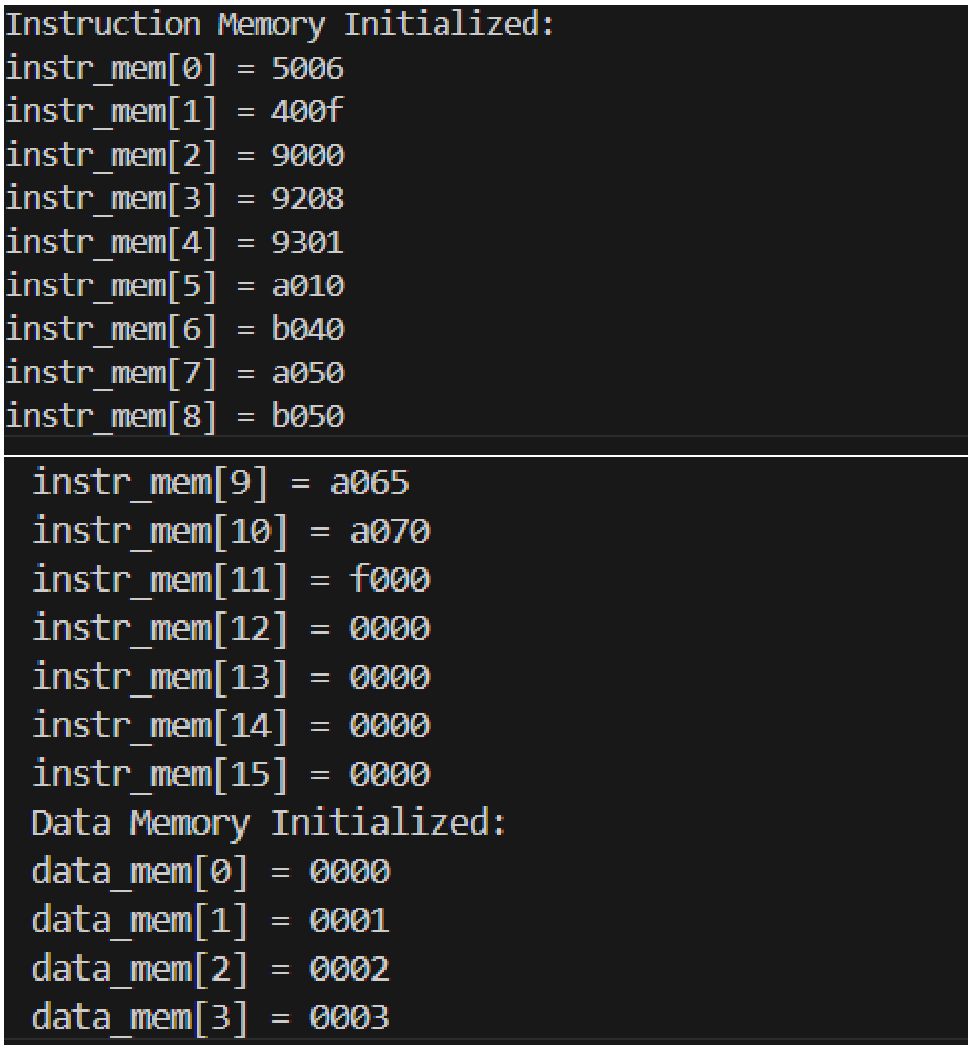
13.2 compute_core_tb.v

Testbench for Compute Core

Compute Core Testbench Code

```
// compute_core_tb.v


module compute_core_tb;

    // Parameters
    parameter DATA_WIDTH = 32;       // V    d   cho  c h i u
        r ng   d    l i u
    parameter NUM_THREADS = 4;       //  S    l    ng  threads
    parameter REG_COUNT = 16;        //  S    l    ng  thanh ghi
       trong  m i  thread
    parameter ADDR_WIDTH = 4;        //  C h i u   r ng     a
        c h   b   n h

    // Inputs
    reg clk;
    reg reset;

    // Outputs
    wire halt;

    // Instantiate the compute_core module
    compute_core uut (
        .clk(clk),
        .reset(reset),
        .halt(halt)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;  //  T o  xung   ng    h    v i  chu
            k   10   n    v    t h i  gian
    end

    // Testbench stimulus
    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;  //  B t    u    v i  reset

        // Display header
        $display("Time\tReset\tHalt\tPC[0]\tPC[1]\tPC[2]\tPC[3]");

        // Apply reset and release it after some time
        #10 reset = 0; //  T h c   h i n  reset  t i   t h i
              im    t = 0

        // Wait for some clock cycles and observe behavior
        #50; //   i    m t   t h i  gian

        // Test - simulate scheduling and instruction execution
        $display("Running computation...");
        #100; //   i    m t  ch t       quan  s  t  h o t
              ng

        // Apply a reset again to see the reset behavior
        reset = 1;
        #10 reset = 0;
        #50;

        // End simulation
        $finish;
    end

    // Monitor the PC and halt signal for debugging
    always @(posedge clk) begin
        $display("%0t\t%0b\t%0b\t%h\t%h\t%h\t%h", $time, reset,
           halt,
                 uut.pc[0], uut.pc[1], uut.pc[2], uut.pc[3]);
    end

endmodule
```

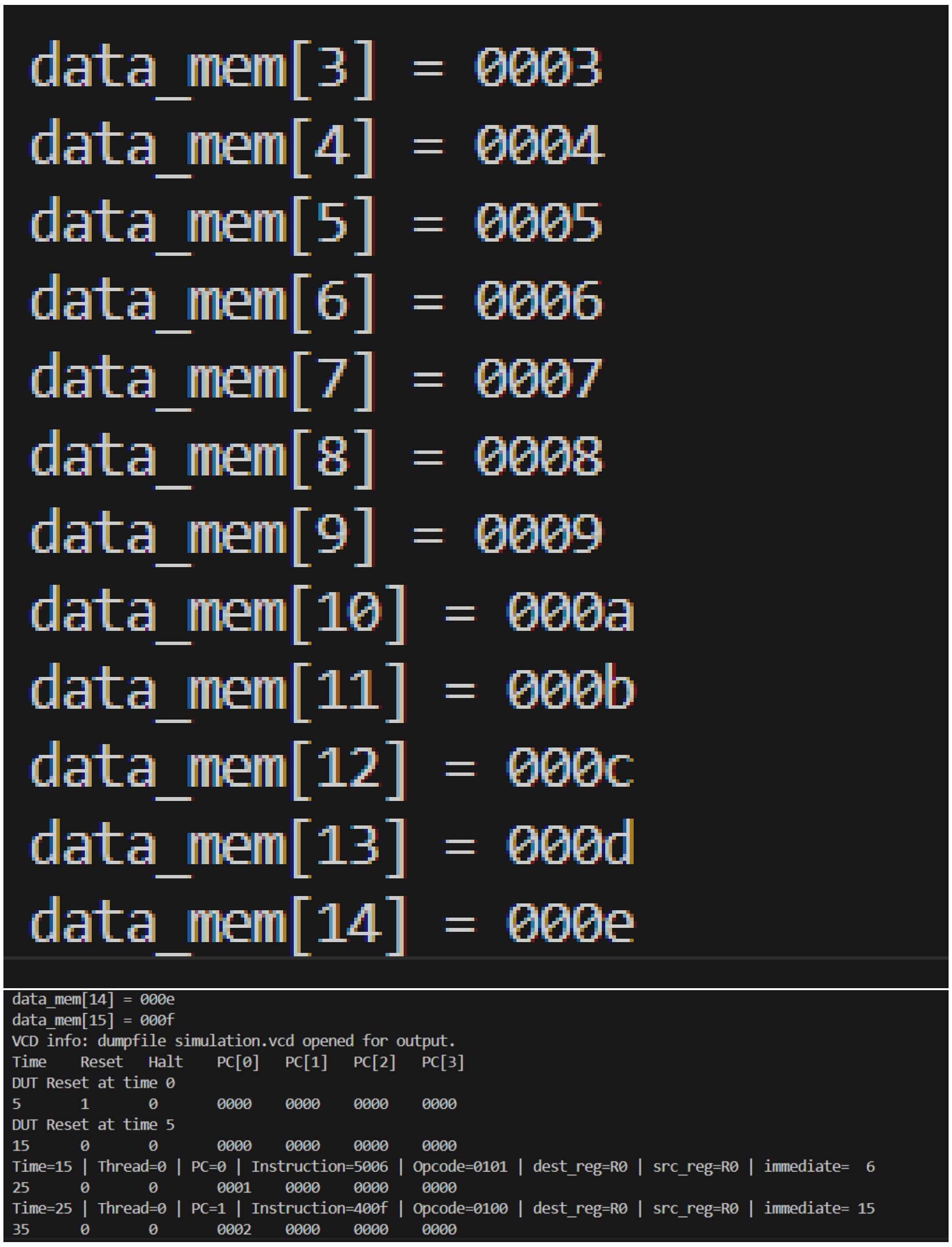
13.3 Result.v



```
Instruction Memory Initialized:
instr_mem[0] = 5006
instr_mem[1] = 400f
instr_mem[2] = 9000
instr_mem[3] = 9208
instr_mem[4] = 9301
instr_mem[5] = a010
instr_mem[6] = b040
instr_mem[7] = a050
instr_mem[8] = b050
```

```
instr_mem[9] = a065
instr_mem[10] = a070
instr_mem[11] = f000
instr_mem[12] = 0000
instr_mem[13] = 0000
instr_mem[14] = 0000
instr_mem[15] = 0000
Data Memory Initialized:
data_mem[0] = 0000
data_mem[1] = 0001
data_mem[2] = 0002
data_mem[3] = 0003
```

Figure 7: Testbench Verilog Output



```
data_mem[3] = 0003
data_mem[4] = 0004
data_mem[5] = 0005
data_mem[6] = 0006
data_mem[7] = 0007
data_mem[8] = 0008
data_mem[9] = 0009
data_mem[10] = 000a
data_mem[11] = 000b
data_mem[12] = 000c
data_mem[13] = 000d
data_mem[14] = 000e
```

```
data_mem[14] = 000e
data_mem[15] = 000f
VCD info: dumpfile simulation.vcd opened for output.
Time    Reset   Halt    PC[0]   PC[1]   PC[2]   PC[3]
DUT Reset at time 0
5       1       0       0000    0000    0000    0000
DUT Reset at time 5
15      0       0       0000    0000    0000    0000
Time=15 | Thread=0 | PC=0 | Instruction=5006 | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate=  6
25      0       0       0001    0000    0000    0000
Time=25 | Thread=0 | PC=1 | Instruction=400f | Opcode=0100 | dest_reg=R0 | src_reg=R0 | immediate= 15
35      0       0       0002    0000    0000    0000
```

Figure 8: Testbench Verilog Output

Predefined Program for Testing Simulation

This program demonstrates the GPU's ability to execute instructions on different threads. The initial memory setup and instructions are as follows:

Memory Setup

- Data Memory:
  - `memory[0] = 3`
  - `memory[5] = 5`
- Instruction Memory: Defined as binary instructions detailed below.

Instructions

Address	Assembly Code	Binary Instruction	Explanation
0x0000	LDR R0, [0]	0x6000	Load value from memory[0] into register R0.
0x0001	LDR R1, [1]	0x6101	Load value from memory[1] into register R1.
0x0002	ADD R2 = R0 + R1 + 16	0x0210	R2 = 3 + 5 + 16 = 24.
0x0003	SUB R3 = R2 - R0 - 0	0x1300	R3 = 24 - 3 = 21.
0x0004	CMP R3, R2	0x3320	Compare R3 and R2. Sets cmp_flag = 1 since R3 < R2.
0x0005	JLT 0x0A	0x500A	Jump to address 0x0A if cmp_flag = 1.
0x0006	MUL R4 = R2 * R3	0x2430	(Only if no jump occurs): R4 = 24 * 21 = 504.
0x0007	STR R4, [2]	0x7402	Store value of R4 into memory[2].
0x000A	LDR R5, [1]	0x6501	Load value from memory[1] into register R5.
0x000B	ADD R5 = R5 + R0 + 0	0x0500	R5 = 5 + 3 + 0 = 8.
0x000C	STR R5, [3]	0x7503	Store value of R5 into memory[3].
0x000D	HALT	0xF000	Halt execution.
0x000E	NOP	0x0000	No operation.
0x000F	NOP	0x0000	No operation.
0x0009	NOP	0x0000	No operation.

Program Runtime Result

The following is the raw output of the program execution:

Program Execution Output

```
.venv  GPUwVerilog git:(main)  make
rm -f results.xml
"/Applications/Xcode.app/Contents/Developer/usr/bin/make" -f Makefile results.xml
rm -f results.xml
MODULE=testbench.test_execution TESTCASE= TOPLEVEL=compute_core TOPLEVEL_LANG=verilog \
         /opt/homebrew/bin/vvp -M /Volumes/FreeSpace/CS365/GPUwVerilog/.venv/
         lib/python3.13/site-packages/cocotb/libs -m libcocotbvpi_icarus   sim_build/sim.vvp
     -.--ns INFO     gpi
     ..mbed/gpi_embed.cpp:109  in set_program_name_in_venv        Using Python virtual environment
     interpreter at /Users/ngotrung/CS365/GPUwVerilog/.venv/bin/python
     -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:101
     in gpi_print_registered_impl       VPI registered
     0.00ns INFO     cocotb                             Running on Icarus Verilog version 12.0 (stable)
     0.00ns INFO     cocotb                             Running tests with cocotb v1.9.2 from
     /Users/ngotrung/CS365/GPUwVerilog/.venv/lib/python3.13/site-packages/cocotb
     0.00ns INFO     cocotb                             Seeding Python random module with 1734276819
     0.00ns INFO     cocotb.regression                  Found test testbench.test_execution.test_executio
     0.00ns INFO     cocotb.regression                  running test_execution (1/1)
     0.00ns INFO     cocotb                             Clock started with 10 ns period. Asserting reset.
Instruction Memory Initialized:
instr_mem[0] = 6000
instr_mem[1] = 6101
instr_mem[2] = 0210
instr_mem[3] = 1300
instr_mem[4] = 3320
instr_mem[5] = 500a
instr_mem[6] = 2430
instr_mem[7] = 7402
instr_mem[8] = f000
instr_mem[9] = 0000
instr_mem[10] = 6501
instr_mem[11] = 0500
instr_mem[12] = 7503
instr_mem[13] = f000
instr_mem[14] = 0000
instr_mem[15] = 0000
Data Memory Initialized:
data_mem[0] = 0003
data_mem[1] = 0005
data_mem[2] = 0008
data_mem[3] = 0007
data_mem[4] = 0000
data_mem[5] = 0000
data_mem[6] = 0000
data_mem[7] = 0000
data_mem[8] = 0000
data_mem[9] = 0000
data_mem[10] = 0000
data_mem[11] = 0000
data_mem[12] = 0000
data_mem[13] = 0000
data_mem[14] = 0000
data_mem[15] = 0000
VCD info: dumpfile simulation.vcd opened for output.
     0.00ns INFO     cocotb                             Deasserting reset.
DUT Reset at time 0
    10.00ns INFO     cocotb                             Waiting for 'halt' signal.
Time=10000 | Thread=0 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=20000 | Thread=0 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=30000 | Thread=1 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=40000 | Thread=2 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=50000 | Thread=3 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=60000 | Thread=4 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=70000 | Thread=5 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=80000 | Thread=6 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=90000 | Thread=7 | PC=0 | Instruction=6000 | Opcode=0110 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=100000 | Thread=0 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=110000 | Thread=1 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=120000 | Thread=2 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
```

```
Time=130000 | Thread=3 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=140000 | Thread=4 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=150000 | Thread=5 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=160000 | Thread=6 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=170000 | Thread=7 | PC=1 | Instruction=6101 | Opcode=0110 | dest_reg=R1 | src_reg=R0 | immediate=  1
Time=180000 | Thread=0 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=190000 | Thread=1 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=200000 | Thread=2 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=210000 | Thread=3 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=220000 | Thread=4 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=230000 | Thread=5 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=240000 | Thread=6 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=250000 | Thread=7 | PC=2 | Instruction=0210 | Opcode=0000 | dest_reg=R2 | src_reg=R1 | immediate= 16
Time=260000 | Thread=0 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=270000 | Thread=1 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=280000 | Thread=2 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=290000 | Thread=3 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=300000 | Thread=4 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=310000 | Thread=5 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=320000 | Thread=6 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=330000 | Thread=7 | PC=3 | Instruction=1300 | Opcode=0001 | dest_reg=R3 | src_reg=R0 | immediate=  0
Time=340000 | Thread=0 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=350000 | Thread=1 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=360000 | Thread=2 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=370000 | Thread=3 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=380000 | Thread=4 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=390000 | Thread=5 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=400000 | Thread=6 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=410000 | Thread=7 | PC=4 | Instruction=3320 | Opcode=0011 | dest_reg=R3 | src_reg=R2 | immediate= 32
Time=420000 | Thread=0 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=430000 | Thread=1 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=440000 | Thread=2 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=450000 | Thread=3 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=460000 | Thread=4 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=470000 | Thread=5 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=480000 | Thread=6 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=490000 | Thread=7 | PC=5 | Instruction=500a | Opcode=0101 | dest_reg=R0 | src_reg=R0 | immediate= 10
Time=500000 | Thread=0 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=510000 | Thread=1 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=520000 | Thread=2 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=530000 | Thread=3 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=540000 | Thread=4 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=550000 | Thread=5 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=560000 | Thread=6 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Time=570000 | Thread=7 | PC=6 | Instruction=2430 | Opcode=0010 | dest_reg=R4 | src_reg=R3 | immediate= 48
Thread 0 executing HALT. Halting.
Time=580000 | Thread=0 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Time=590000 | Thread=1 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=600000 | Thread=2 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=610000 | Thread=3 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=620000 | Thread=4 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=630000 | Thread=5 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=640000 | Thread=6 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Time=650000 | Thread=7 | PC=7 | Instruction=7402 | Opcode=0111 | dest_reg=R4 | src_reg=R0 | immediate=  2
Thread 1 executing HALT. Halting.
Time=660000 | Thread=1 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Thread 2 executing HALT. Halting.
Time=670000 | Thread=2 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Thread 3 executing HALT. Halting.
Time=680000 | Thread=3 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Thread 4 executing HALT. Halting.
Time=690000 | Thread=4 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Thread 5 executing HALT. Halting.
Time=700000 | Thread=5 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Thread 6 executing HALT. Halting.
Time=710000 | Thread=6 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
Thread 7 executing HALT. Halting.
Time=720000 | Thread=7 | PC=8 | Instruction=f000 | Opcode=1111 | dest_reg=R0 | src_reg=R0 | immediate=  0
All threads have halted at time 720000
   730.00ns INFO     cocotb                              Test PASSED: 'halt' signal asserted after 72
   cycles.
   730.00ns INFO     cocotb.regression                   test_execution passed
   730.00ns INFO     cocotb.regression
   ******************************************************************************************
** TEST                                    STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*********************************************************************************************
** testbench.test_execution.test_execution   PASS          730.00           0.01     103738.65  **
*********************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0                            730.00           0.01      75599.27  **
*********************************************************************************************
```

Development Progress Documentation

- Week 1: Project initiation, GPU architecture research, and team task allocation.
- Week 2: Verilog syntax learning and setting up the development environment.
- Week 3-4: High-level architecture design and initial module specifications.
- Week 5-6-7: Module development
- Week 7: Integration and simulation of the predefined program.
- Week 8: Debugging and optimization of components.

Contribution Segment

Task Distribution Table

Member	Task Weight (%)	Details of Contribution
1. Ngo Thanh Trung	24%	Implemented `fetcher.v`, `scheduler.v`, the computer core unit `computer_core.v`, and `definitions.vh`. Initialized pre-defined program `data_memory.mem`, `instruction_memory.mem`, the Makefile for the program, and contributed to the register file implementation. Developed the cocotb test bench for managing the program run time successness.
3. Pham Tien Dat	19%	Designed the instruction decoder `decoder.v`, developed the ALU `alu.v`, and analyzed simulation results for correctness. Designed the architecture diagram along with Huy.
2. Le Quang Huy	19%	Conducted research on GPU architecture, contributed to the pre-defined program design and architecture diagram design, supported the development of project documentation.
4. Bui Dang Quoc An	19%	Created test benches for all modules along with Khoi. Ensured their outputs compatibility with predefined programs.
5. Nguyen Duy Khoi	19%	Developed the memory interface module and created test benches for all modules.

Table 1: Task Distribution Table

Contribution Ranking Table

Rank	Member Name
1	Ngo Thanh Trung
2	Pham Tien Dat
2	Le Quang Huy
2	Bui Dang Quoc An
2	Nguyen Duy Khoi

Acknowledgements

We have our deepest gratitude to Dr. Nguyen Dinh Han for his invaluable lectures that guided us in development of this project. We would also like to thank open-source contributors like `adam-maj/tiny-gpu` for inspiring our work.