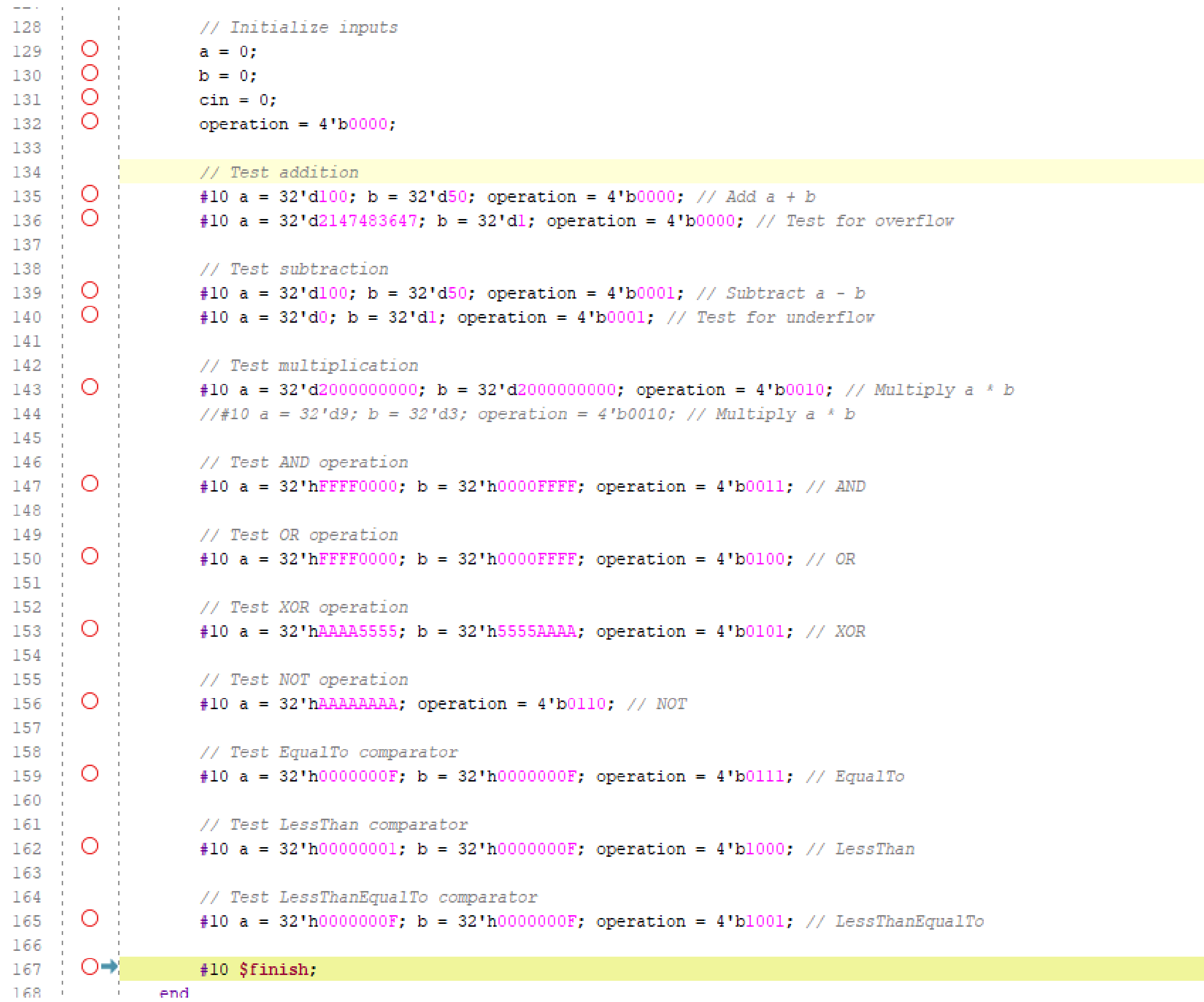
Mini Project

Pranav Konduru | Jose Guerra Fajardo

ALU

- Following test cases:

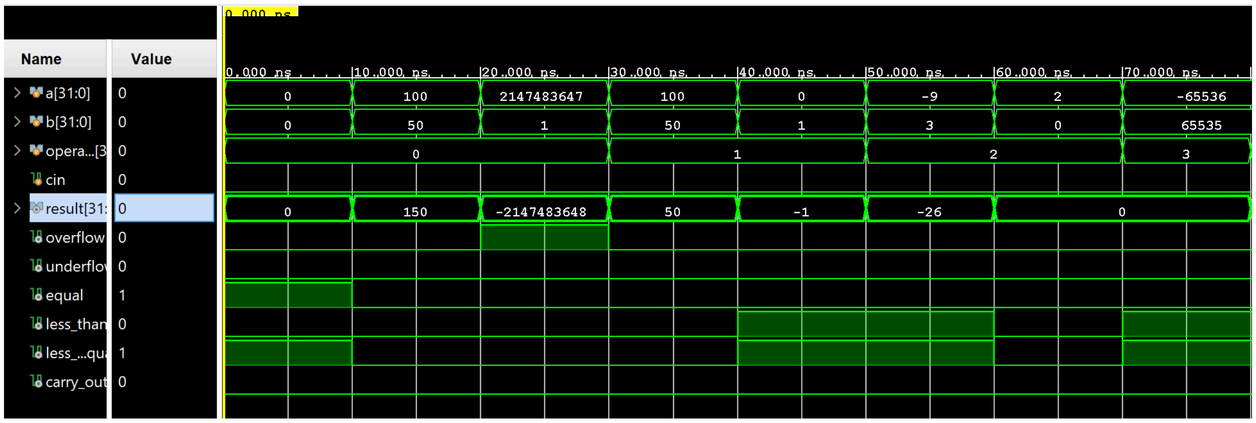


```
        // Initialize inputs
        a = 0;
        b = 0;
        cin = 0;
        operation = 4'b0000;

        // Test addition
        #10 a = 32'd100; b = 32'd50; operation = 4'b0000; // Add a + b
        #10 a = 32'd2147483647; b = 32'd1; operation = 4'b0000; // Test for overflow

        // Test subtraction
        #10 a = 32'd100; b = 32'd50; operation = 4'b0001; // Subtract a - b
        #10 a = 32'd0; b = 32'd1; operation = 4'b0001; // Test for underflow

        // Test multiplication
        #10 a = 32'd2000000000; b = 32'd2000000000; operation = 4'b0010; // Multiply a * b
        //#10 a = 32'd9; b = 32'd3; operation = 4'b0010; // Multiply a * b

        // Test AND operation
        #10 a = 32'hFFFF0000; b = 32'h0000FFFF; operation = 4'b0011; // AND

        // Test OR operation
        #10 a = 32'hFFFF0000; b = 32'h0000FFFF; operation = 4'b0100; // OR

        // Test XOR operation
        #10 a = 32'hAAAA5555; b = 32'h5555AAAA; operation = 4'b0101; // XOR

        // Test NOT operation
        #10 a = 32'hAAAAAAAA; operation = 4'b0110; // NOT

        // Test EqualTo comparator
        #10 a = 32'h0000000F; b = 32'h0000000F; operation = 4'b0111; // EqualTo

        // Test LessThan comparator
        #10 a = 32'h00000001; b = 32'h0000000F; operation = 4'b1000; // LessThan

        // Test LessThanEqualTo comparator
        #10 a = 32'h0000000F; b = 32'h0000000F; operation = 4'b1001; // LessThanEqualTo

        #10 $finish;
    end
```

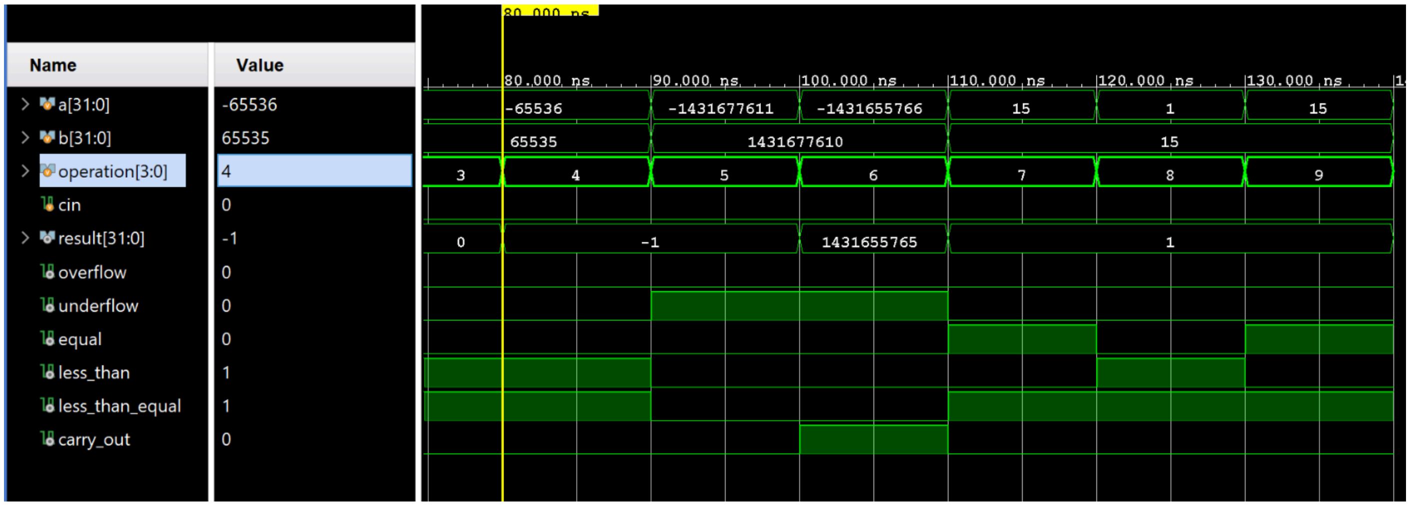
-
- Inputs are signed 32 bit integers and the opcode used to select the ALU operation. Outputs are 32 bit results and flag bits for overflow, underflow, and the rest of the comparator flags

Mnemonic	Opcode (4-bit)	Description
`ADD`	`0000`	Add two registers
`SUB`	`0001`	Subtract two registers
`MUL`	`0010`	Multiply two registers
`AND`	`0011`	AND operation between registers
`OR`	`0100`	OR operation between registers
`XOR`	`0101`	XOR operation between registers
`NOT`	`0110`	NOT operation on a register
`CMP_EQ` (EqualTo)	`0111`	Compare Equal
`CMP_LT` (LessThan)	`1000`	Compare Less Than
`CMP_LTE` (LessThanEqualTo)	`1001`	Compare Less Than or Equal
`LOAD`	`1010`	Load data from memory
`STORE`	`1011`	Store data to memory
`JMP`	`1100`	Unconditional jump to register
`JMP_COND`	`1101`	Conditional jump based on CC
`CALL`	`1110`	Call a procedure (save state)
`RET`	`1111`	Return from procedure (restore state)

```
    assign opcode = instruction[31:28];  // 4-bit opcode
    assign reg1 = instruction[27:24];    // Source register 1
    assign reg2 = instruction[23:20];    // Source register 2
    assign reg_dest = instruction[19:16]; // Destination register
```



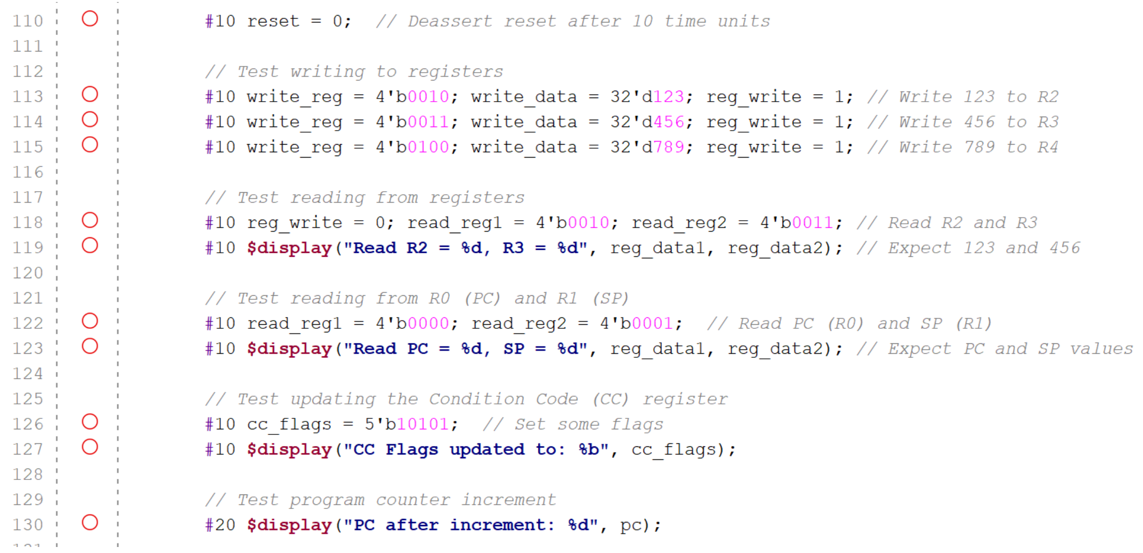
Waveform that shows the results for the first four operations: ADD, SUB, MULT, AND



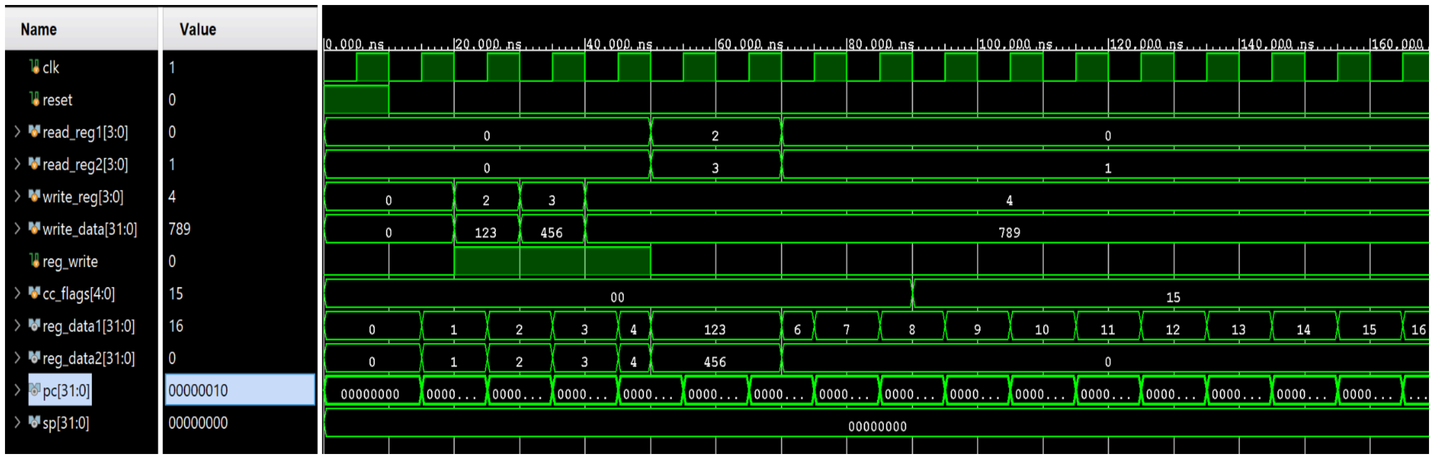
Waveform that shows the results for the first four operations: OR, XOR, NOT, Equal, Less Than, Less Than Equal To

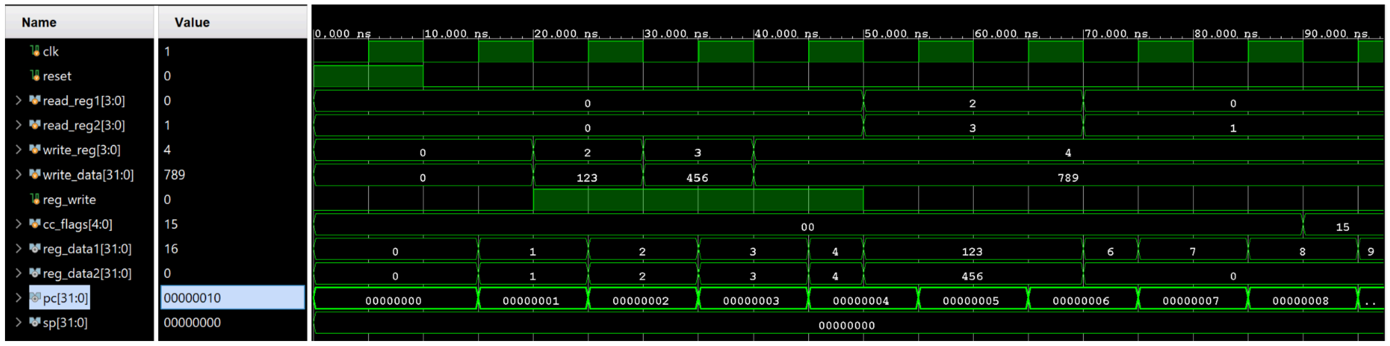
Registers

- Tested reading and writing to the 16 32-bit register files by feeding values along with the register location.
- Setup a write flag condition where data can only be written once the signal (write flag) is high
- Setup a condition for the PC and SP so that data fed to their designated address will go to their assigned register.



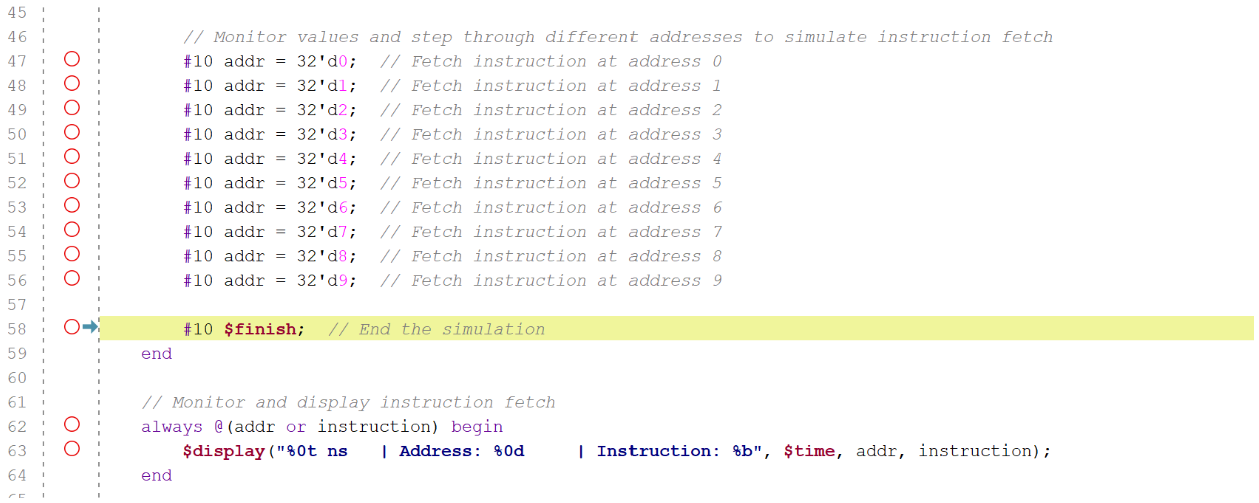
```
        #10 reset = 0;  // Deassert reset after 10 time units

        // Test writing to registers
        #10 write_reg = 4'b0010; write_data = 32'd123; reg_write = 1; // Write 123 to R2
        #10 write_reg = 4'b0011; write_data = 32'd456; reg_write = 1; // Write 456 to R3
        #10 write_reg = 4'b0100; write_data = 32'd789; reg_write = 1; // Write 789 to R4

        // Test reading from registers
        #10 reg_write = 0; read_reg1 = 4'b0010; read_reg2 = 4'b0011; // Read R2 and R3
        #10 $display("Read R2 = %d, R3 = %d", reg_data1, reg_data2); // Expect 123 and 456

        // Test reading from R0 (PC) and R1 (SP)
        #10 read_reg1 = 4'b0000; read_reg2 = 4'b0001;  // Read PC (R0) and SP (R1)
        #10 $display("Read PC = %d, SP = %d", reg_data1, reg_data2); // Expect PC and SP values

        // Test updating the Condition Code (CC) register
        #10 cc_flags = 5'b10101;  // Set some flags
        #10 $display("CC Flags updated to: %b", cc_flags);

        // Test program counter increment
        #20 $display("PC after increment: %d", pc);
```





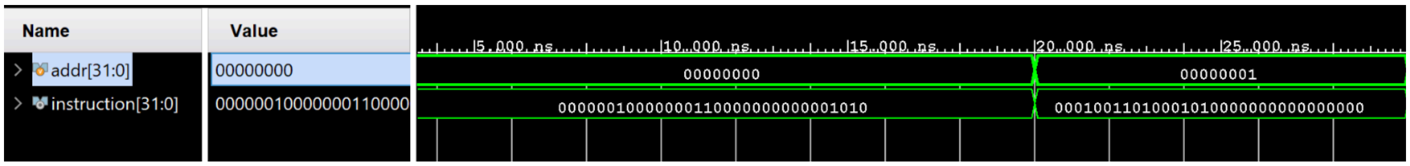
This is the result of running the registerfile module by itself, using a separate test bench. In here we can see how the PC register increments and how the module is working how it's supposed to. We passed some values into the registers to make sure data was being written. We are reading two registers based on the addresses that are being passed(r1,r2).

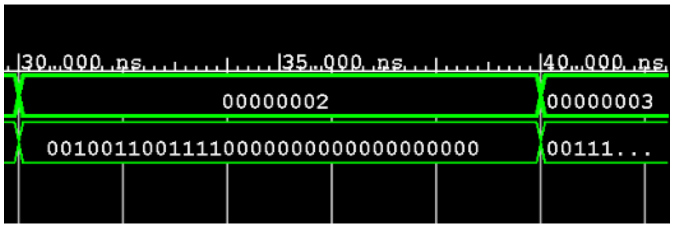
Instruction Memory



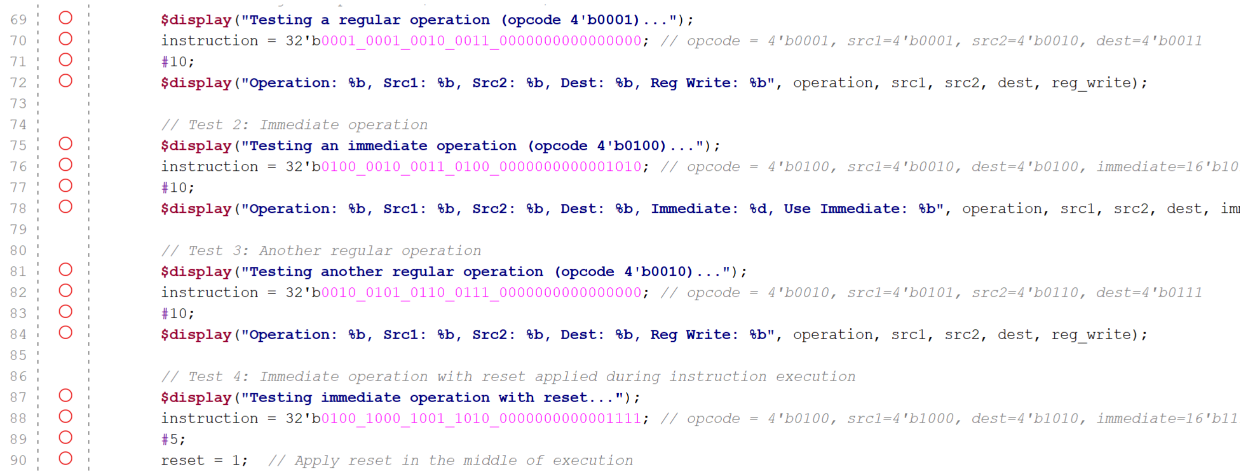
```
        // Monitor values and step through different addresses to simulate instruction fetch
        #10 addr = 32'd0;  // Fetch instruction at address 0
        #10 addr = 32'd1;  // Fetch instruction at address 1
        #10 addr = 32'd2;  // Fetch instruction at address 2
        #10 addr = 32'd3;  // Fetch instruction at address 3
        #10 addr = 32'd4;  // Fetch instruction at address 4
        #10 addr = 32'd5;  // Fetch instruction at address 5
        #10 addr = 32'd6;  // Fetch instruction at address 6
        #10 addr = 32'd7;  // Fetch instruction at address 7
        #10 addr = 32'd8;  // Fetch instruction at address 8
        #10 addr = 32'd9;  // Fetch instruction at address 9

        #10 $finish;  // End the simulation
    end

    // Monitor and display instruction fetch
    always @(addr or instruction) begin
        $display("%0t ns   | Address: %0d     | Instruction: %b", $time, addr, instruction);
    end
```

Input is the memory address and the output is the instruction stored at that address

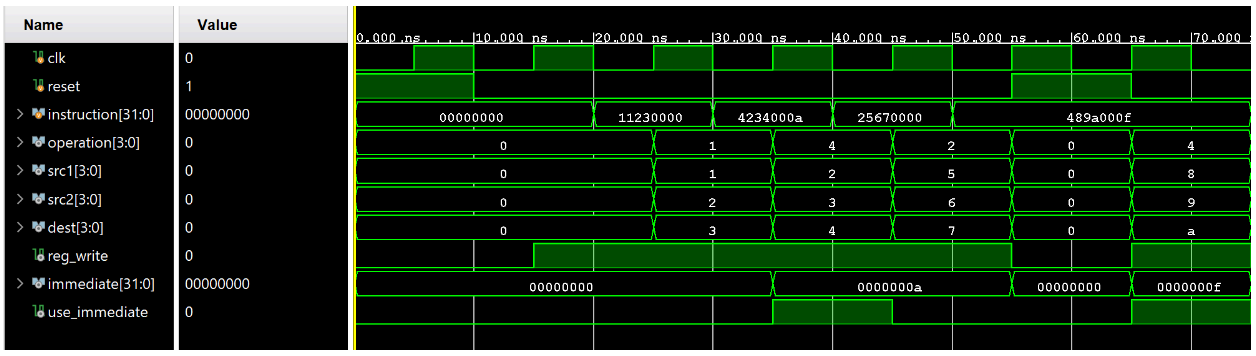




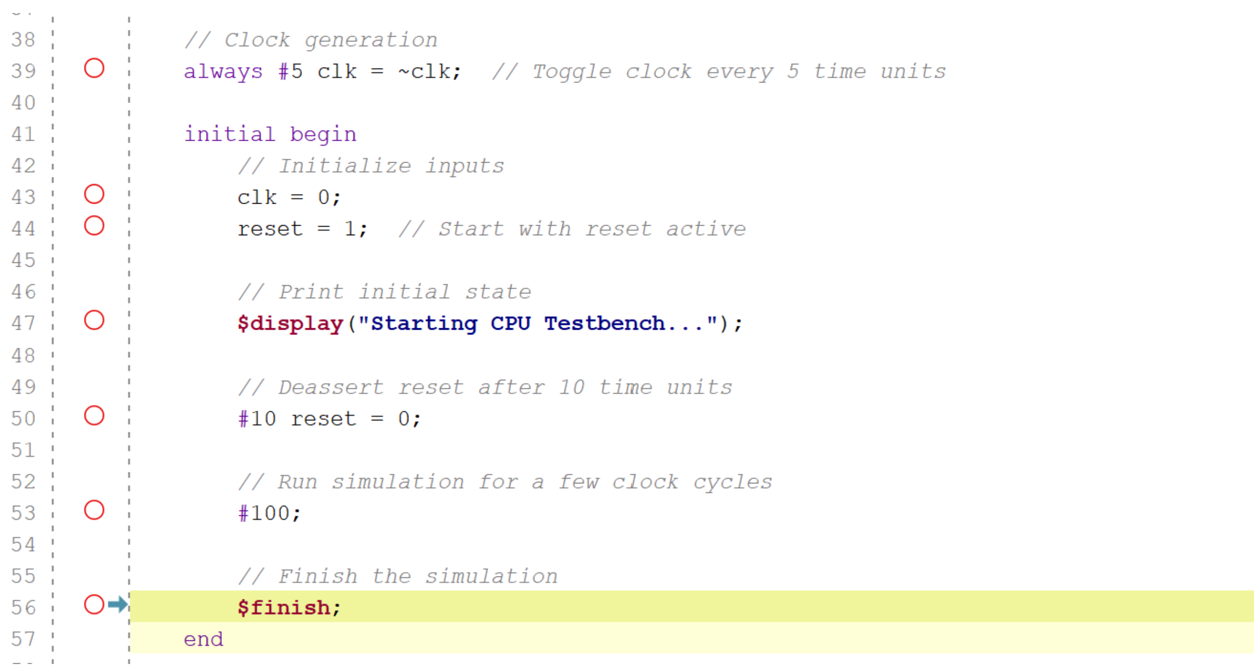
Control Unit



```
        $display("Testing a regular operation (opcode 4'b0001)...");
        instruction = 32'b0001_0001_0010_0011_0000000000000000; // opcode = 4'b0001, src1=4'b0001, src2=4'b0010, dest=4'b0011
        #10;
        $display("Operation: %b, Src1: %b, Src2: %b, Dest: %b, Reg Write: %b", operation, src1, src2, dest, reg_write);

        // Test 2: Immediate operation
        $display("Testing an immediate operation (opcode 4'b0100)...");
        instruction = 32'b0100_0010_0011_0100_0000000000001010; // opcode = 4'b0100, src1=4'b0010, dest=4'b0100, immediate=16'b10
        #10;
        $display("Operation: %b, Src1: %b, Src2: %b, Dest: %b, Immediate: %d, Use Immediate: %b", operation, src1, src2, dest, im

        // Test 3: Another regular operation
        $display("Testing another regular operation (opcode 4'b0010)...");
        instruction = 32'b0010_0101_0110_0111_0000000000000000; // opcode = 4'b0010, src1=4'b0101, src2=4'b0110, dest=4'b0111
        #10;
        $display("Operation: %b, Src1: %b, Src2: %b, Dest: %b, Reg Write: %b", operation, src1, src2, dest, reg_write);

        // Test 4: Immediate operation with reset applied during instruction execution
        $display("Testing immediate operation with reset...");
        instruction = 32'b0100_1000_1001_1010_0000000000001111; // opcode = 4'b0100, src1=4'b1000, dest=4'b1010, immediate=16'b11
        #5;
        reset = 1;  // Apply reset in the middle of execution
```

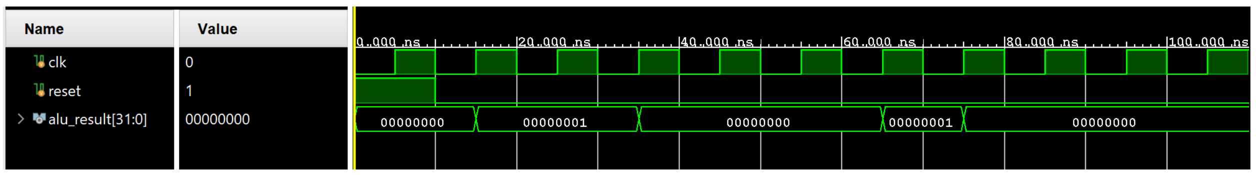


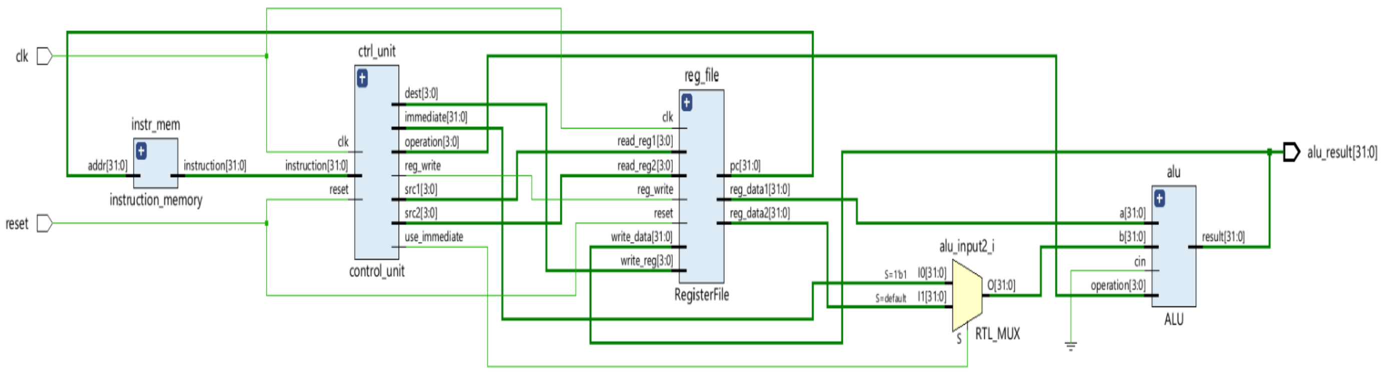
CPU module

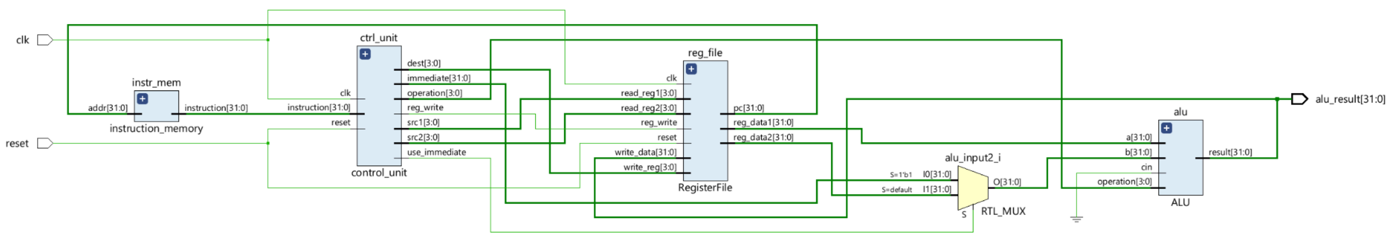


```
    // Clock generation
    always #5 clk = ~clk;  // Toggle clock every 5 time units

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;  // Start with reset active

        // Print initial state
        $display("Starting CPU Testbench...");

        // Deassert reset after 10 time units
        #10 reset = 0;

        // Run simulation for a few clock cycles
        #100;

        // Finish the simulation
        $finish;
    end
```

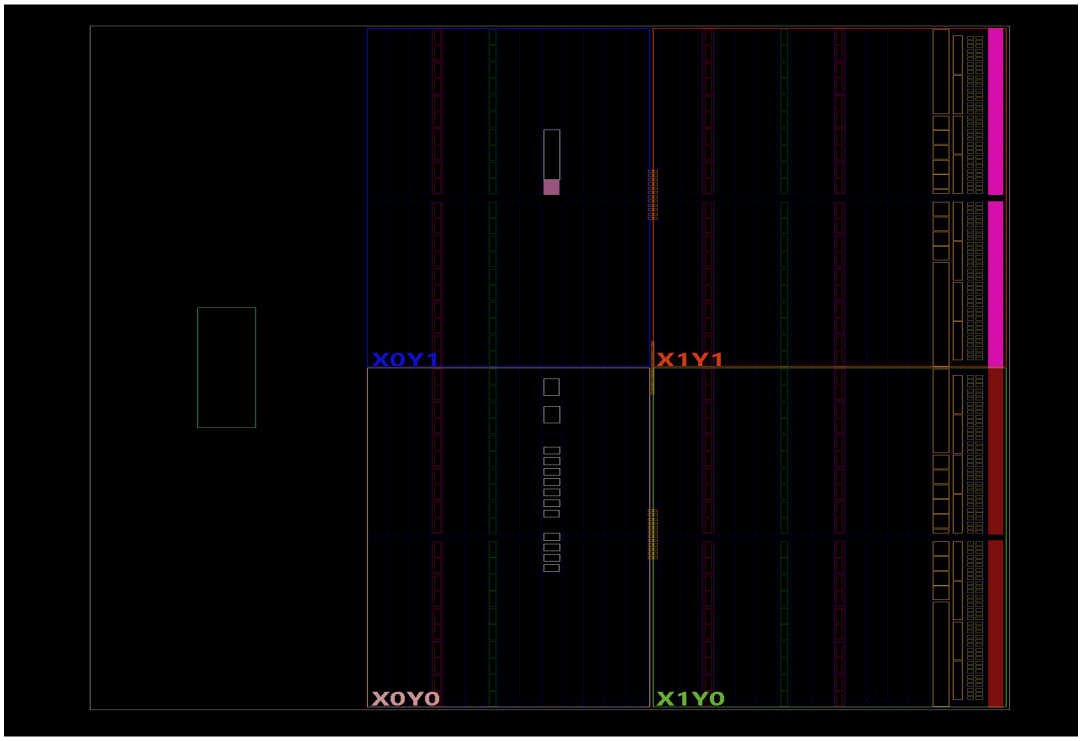
Inputs are clock and reset signal. Output is the result getting stored in the register.







Final schematic of our design.



Final Device

Conclusion

The CPU module design consists of four components: ALU, register file, instruction memory, and control unit. Each of these components were tested with their own testbench and simulated in the waveforms. The overall result, however, did not produce the expected results. Either the instructions are not reaching the control unit correctly or the register file is not updating due to not recognizing the instructions. (We will follow up with these problems and debug them) Although the project does not operate correctly in the end, it helped us understand how instructions are generally passed in a RISC based computer architecture system.