CSE 490 Computer Architecture, Spring 2025

Project 1:
Design of a 16-bit CPU (non-pipelined)

Team 5
Rachel Chan,
Xin Virgil Lu,
Diyun Cheung,
Lucy Bychkov

Table of Content

1. Introduction.....3
2. Detailed Simulation & Hardware Running Instruction..... 3
    2.1. Simulation.....3
    2.2 Hardware Running Instruction..... 5
3. Datapath & Control Path in Elaborated Design Diagram.....9
4. Modules Description.....9
5. What We Have Changed.....21
Reference..... 23
Appendix.....24

1. Introduction

The objective of this project includes implementing, simulating, and synthesizing a simple 16-bit processor. To implement the processor, we will be using softwares such as Xilinx Vivado and Verilog. The 16-bit processor should be designed in a simple, single-cycle, and non-pipelined design. The specific components that will be implemented in the design includes instruction memory, data memory, register file, ALU, sign extension, control unit(s), multiplexer(s), and program counter.

To verify functionality of the simple 16-bit processor and ensure that it is fully functional, we will be checking the functionality in two ways: 1. running a software simulation of the design and 2. create what is known as a "bitstream" to target a Xilinx FPGA board and programming the FPGA board with that bitstream.

2. Detailed Simulation & Hardware Running Instruction

(Note: All the following instruction sets list are based on "demo_program" file provided by TA at demo day, the file can be found here)

2.1. Simulation

First add all source files as well as the simulation testbench files, constraint file; and then "Run Behavioral Simulation", then use the simulation control buttons: 1. Restart, 2. Run 10 us, finally you can get the result below, see Fig 2.

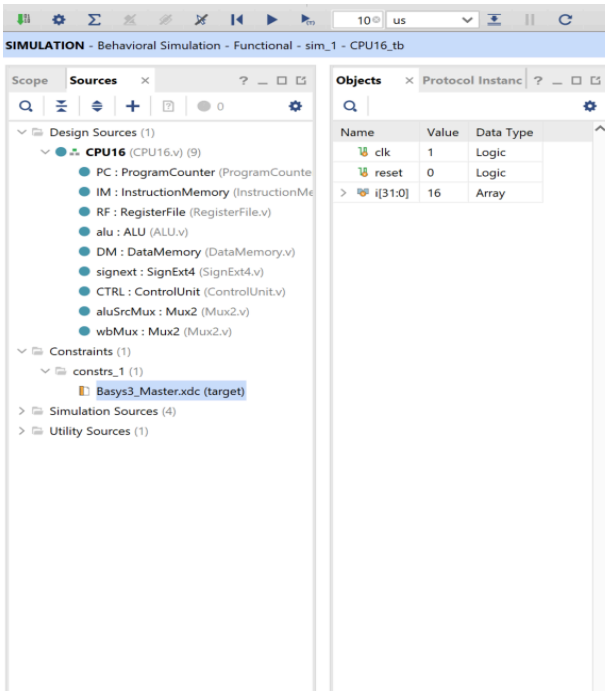


Fig 1. File Structure

In the simulation test bench result shown below, we can focus on the “instr[15:0]”, “opcode[3:0]”, “rt[3:0]”, “rs[3:0]”, “func[3:0]”, “addr[11:0]”. The “rt[3:0]” is the **destination register!**

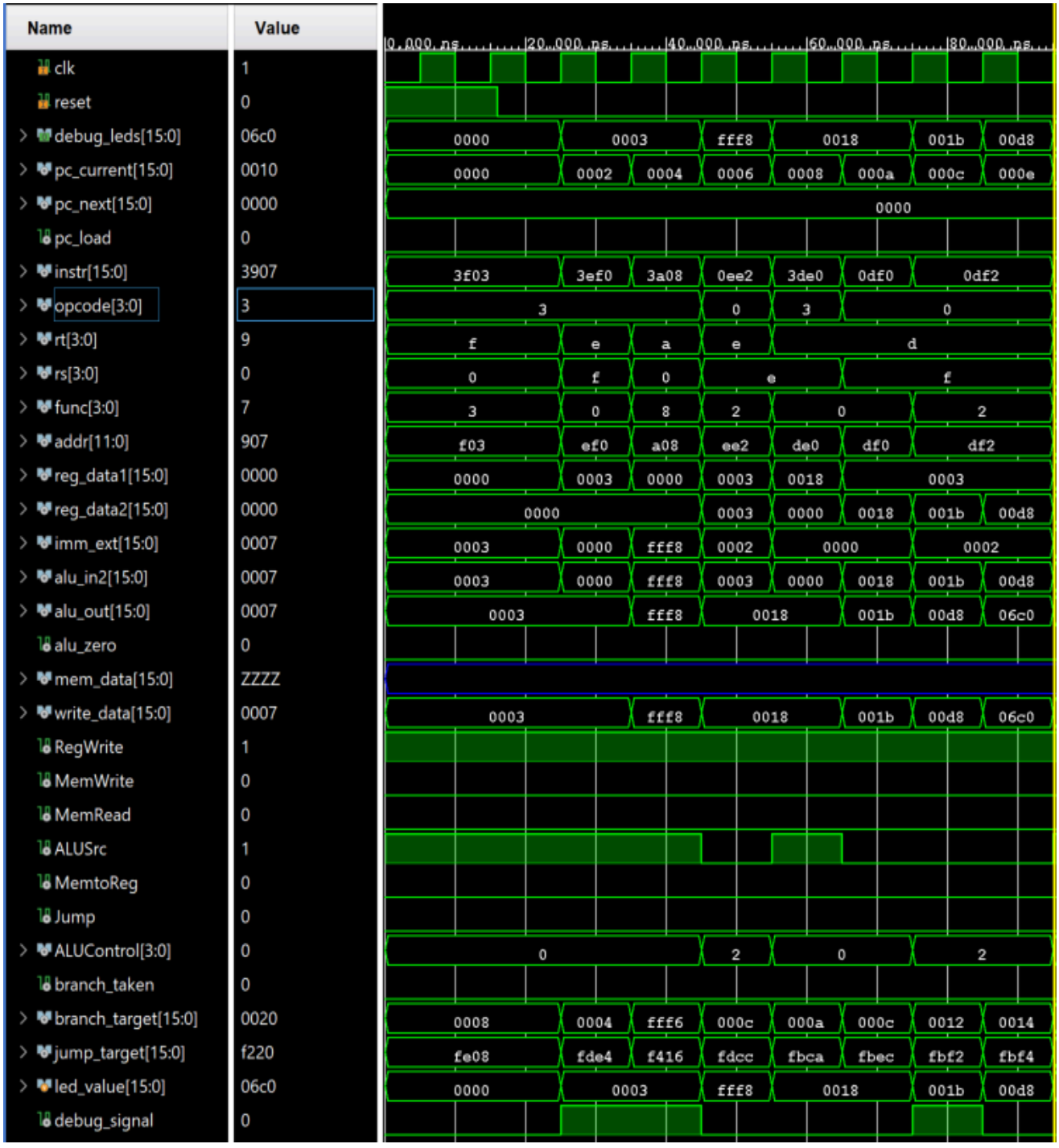


Fig 2. Simulation Result

2.2 Hardware Running Instruction

The 16-bit debug output (debug_leds) is mapped to the 16 onboard LEDs. In our design, these LEDs display the value of the destination register.

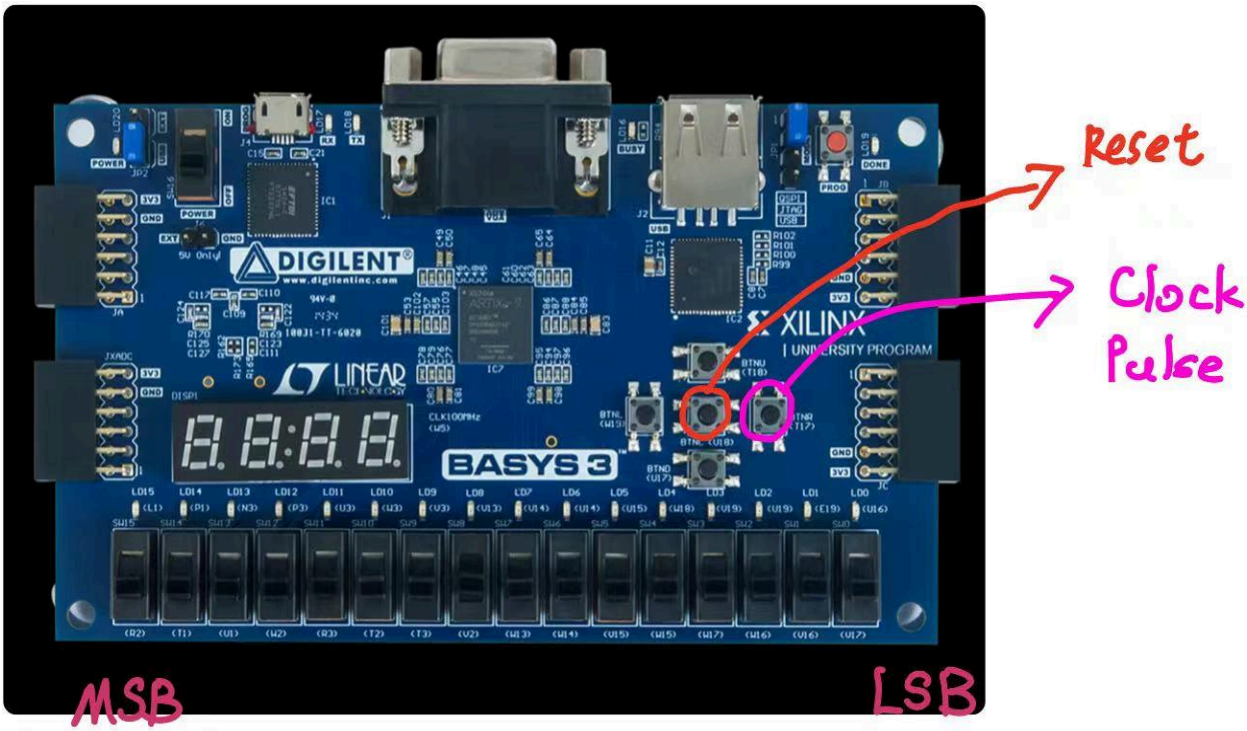


Fig 3. Button Assignment

(The clock input(clk) is connected to the center push button(BTNC) using PACKAGE_PIN T17, as specified in the XDC file; the reset signal(reset) is connected to a designated push button on PACKAGE_PIN U18)

According to the hardware requirements, while holding down the push button (which generates a clock pulse), the new value written into the destination register is immediately captured and remains displayed on the LEDs until the next instruction writes a new value.

I will give a short example because there are so many instructions in the “demo_program.mem”. Program the board with the bitstream; press the center push button to simulate each clock edge(executing one instruction at a time).

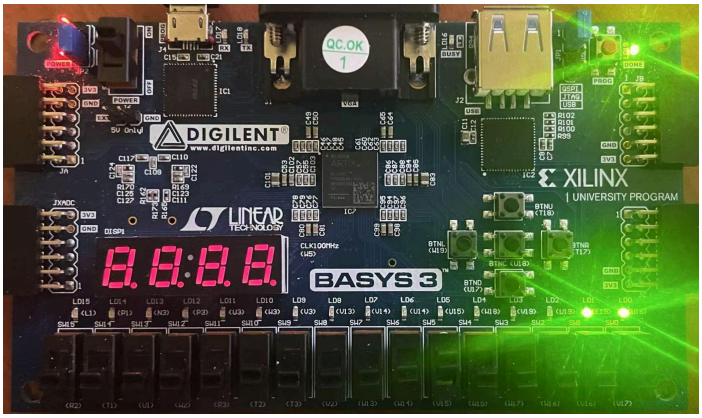


Fig 4.1 After first push
(Instruction: 0011111100000011;
Destination Register value: 3(decimal))

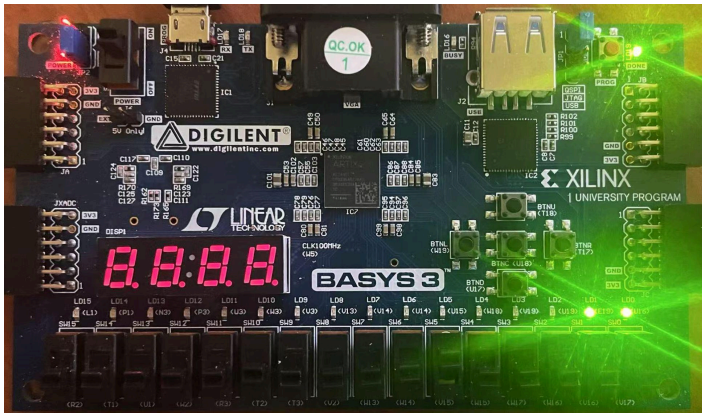


Fig 4.2 After second push
(Instruction: 0011111011110000;
Destination Register value: 3(decimal))

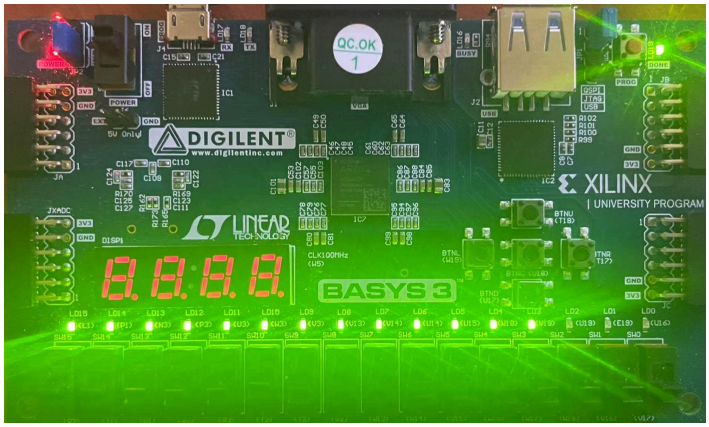


Fig 4.3 After third push
(Instruction: 0011101000001000;
Destination Register value: -8(decimal))

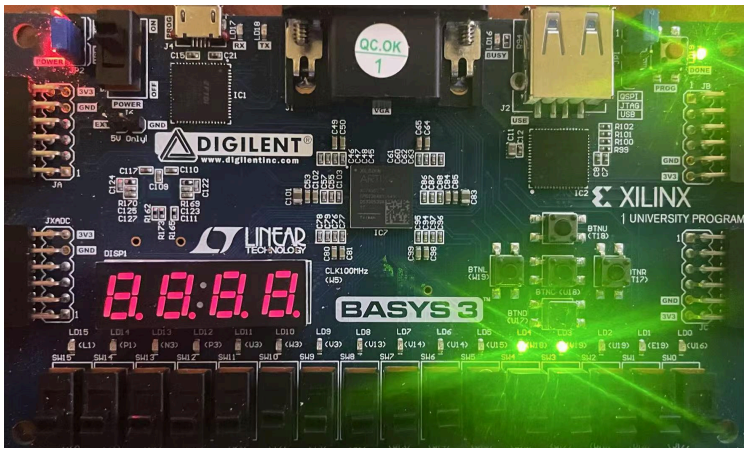


Fig 4.4 After fourth push
(Instruction: 0000111011100010;
Destination Register value: 24(decimal))

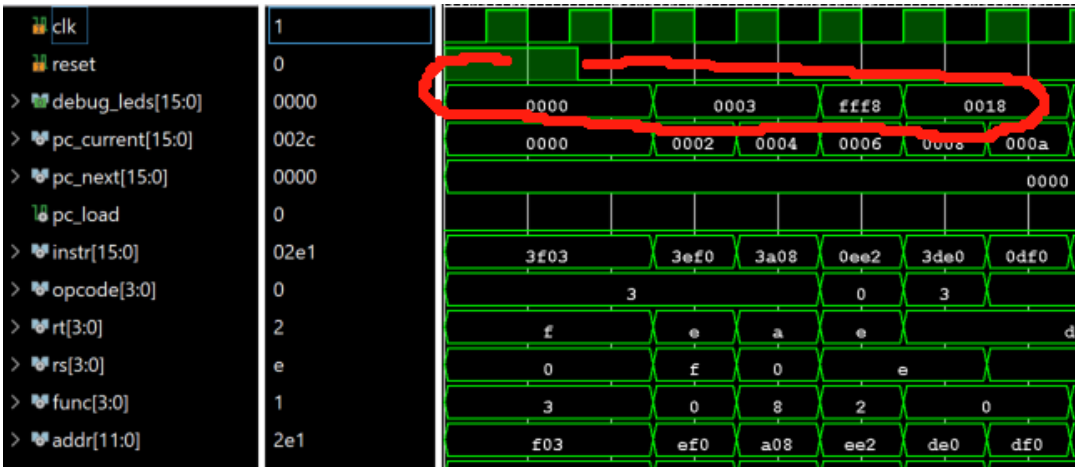


Fig 4.5 Compare to simulation result

The LED has shown the destination register's new value after the instruction completes. And as you can see that all the results above are aligned with expectation.

3. Datapath & Control Path in Elaborated Design Diagram

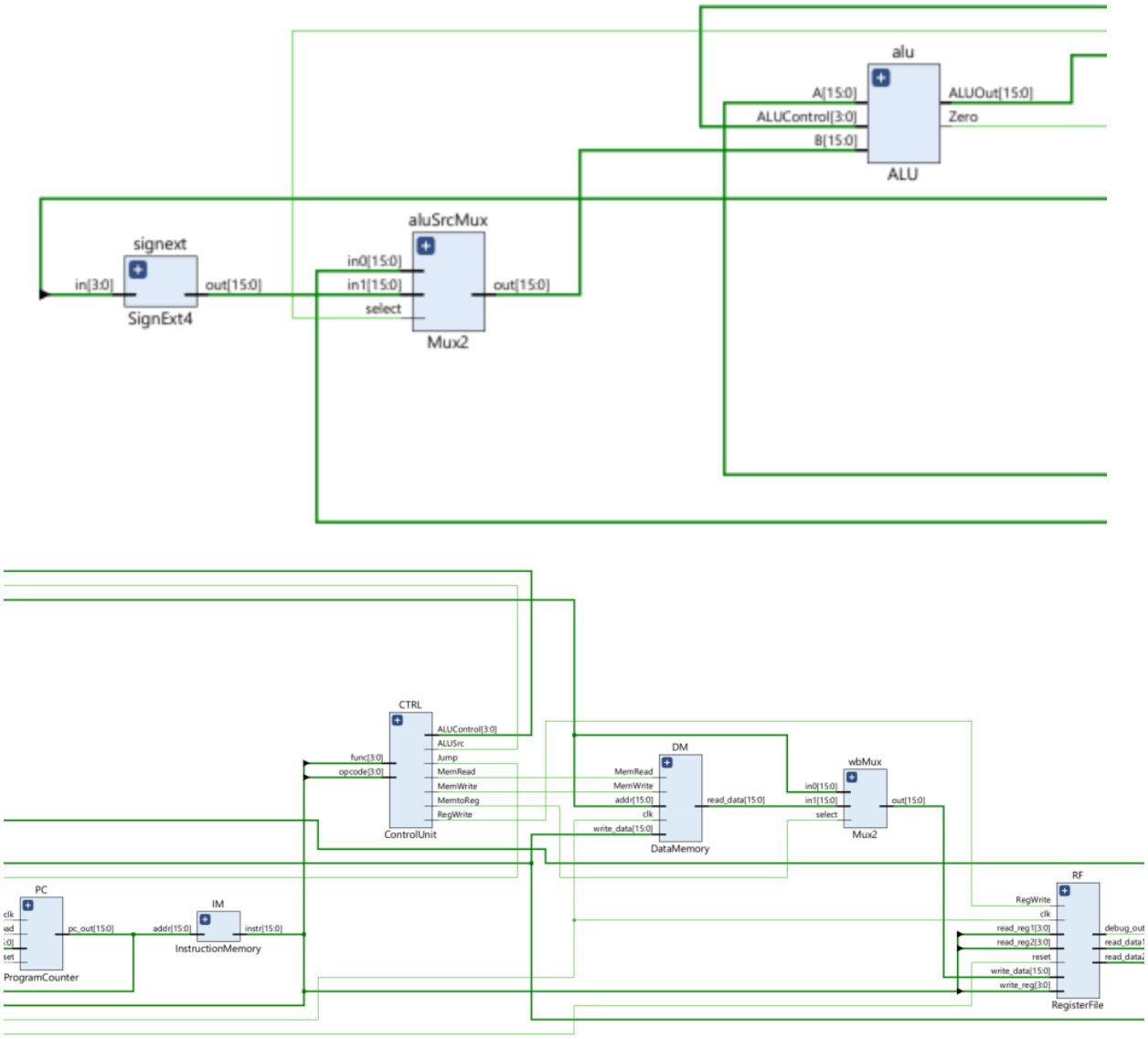


Fig 5. Data Path(Up) & Control Path(Down)

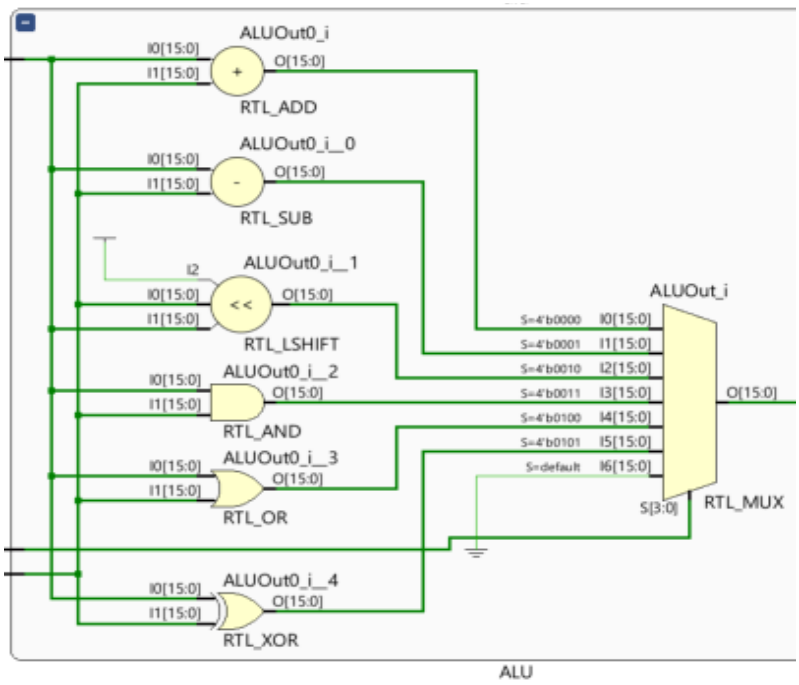
4. Modules Description

Diyun:

- ALU.v:

This module is used to perform basic arithmetic. The ALU takes two 16 bit inputs A and B, along with a 4 bit control signal(ALUcontrol), and produces a 16 bit

result(ALUOut) plus a zero flag. The zero flag is set to 1 whenever the result is 0x0000.




```
A + B;    // add
A - B;    // sub
B << A;   // sll
A & B;    // and
        /* Plan to implement others  */
```

Fig 6. Elaborated Design(Up) & Basic arithmetic operations code(Down)

- ALU_TestBench.v:

The ALU successfully implements the required operations. The ALUControl here is used to select which operation to perform, the value of the ALUControl comes from the output of control_unit, see below and below.The simulation confirms that for various input combinations, the ALU produces the expected result and zero flag.

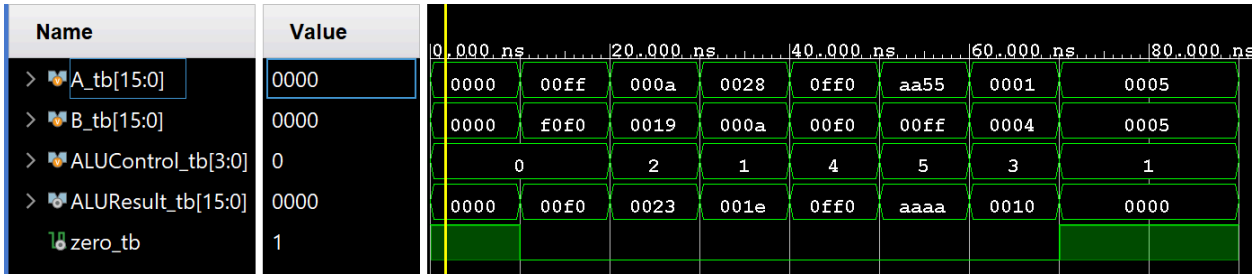
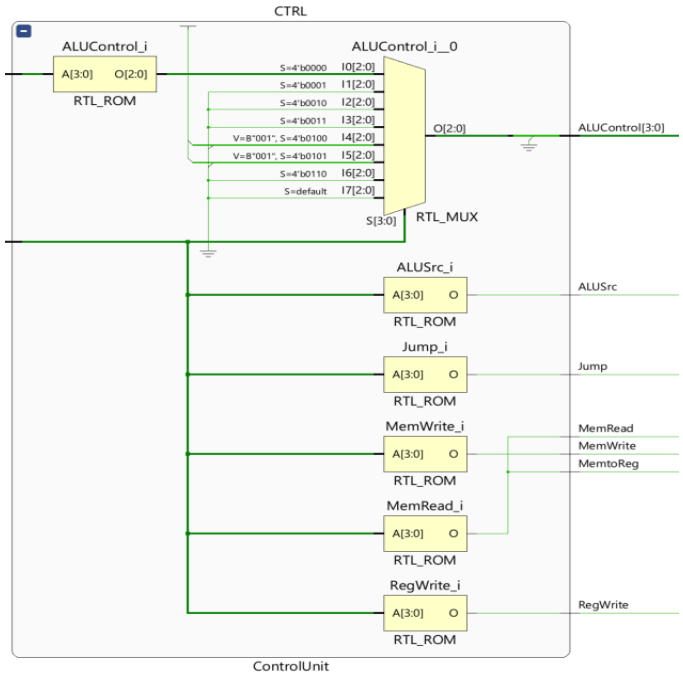


Fig 7. TestBench for ALU

- ControlUnit.v:

This module is to generate the control signals that drive the CPU's datapath. It decodes the opcode and the function field for R-type instructions to produce control signals such as RegWrite, MemWrite, MemRead, ALUSrc, MemtoReg, Jump, and ALUControl. These signals determine: how the register file is accessed; type of memory operation; the arithmetic or logic operation the ALU must perform. I'm thinking about using this module to only control my ALU.v. Below is the block diagram for it.



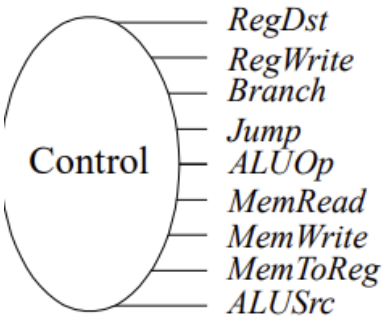


Fig 8. Elaborated Design(Up) & My reference(Down)
(Reference: Computer Organization and Design RIC-V Edition)

- ControlUnit_TestBench.v:

The Control Unit successfully implements the required operations. For each instruction type, the correct control signals are produced. As demonstrated in Fig 4, the simulation output for instructions such as R-type matches the expected signal values.(My simulation testing order: add, sub, sll, and, lw, sw, addi, beq, bne, jmp)

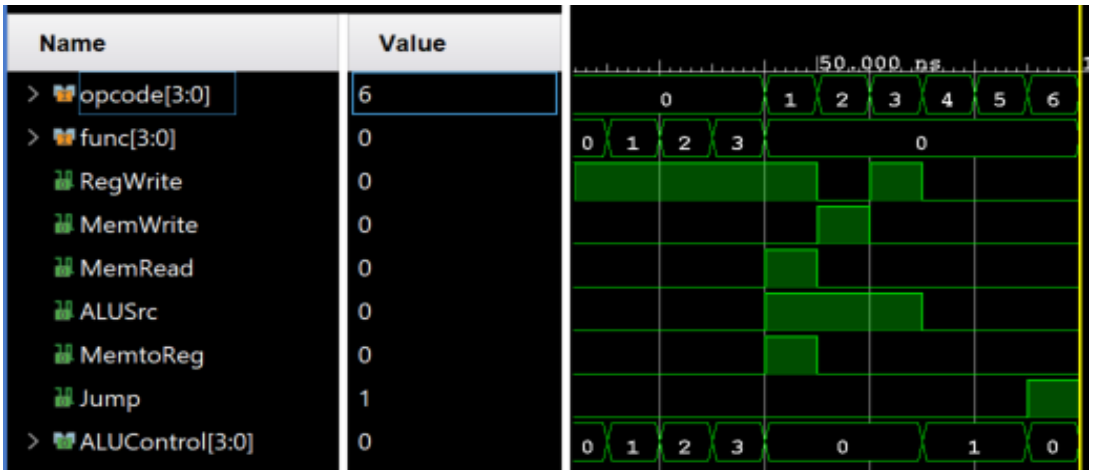


Fig 9. TestBench for ControlUnit

- DataMemory.v:

Our data memory module models a byte-addressable memory(with 128 Bytes) used for load and store operation. Data is stored in big-endian format, meaning that the high byte is stored at the lower address and the low byte at the next address. During the simulation, all memory locations are initially set to 0, fulfilling the requirement for memory initialization.

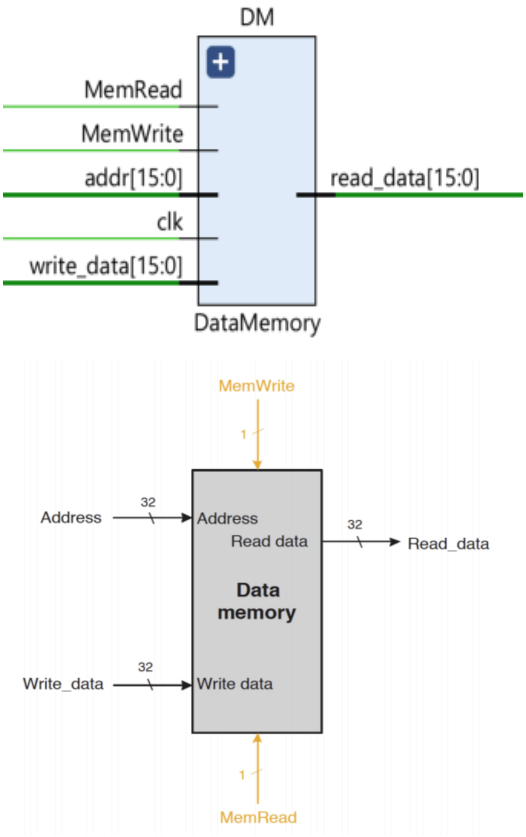


Fig 10. Elaborated Design(Up)& A Simple Data Memory Example(down) ([Reference])

- DataMemory_TestBench.v:

The Data Memory successfully implements the required operations. Our test bench for DataMemory verifies the write and read operations. For example, after writing 0xABCD to a specified address, the test bench reads the same address and confirms that the correct data is retrieved.

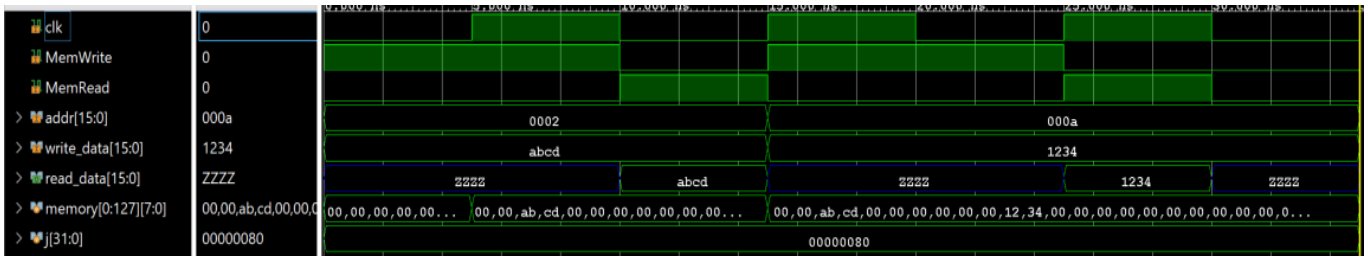


Fig 11. Test Bench for Data Memory

- InstructionMemory.v:

The Instruction Memory module stores the program instructions. In my design, the module uses a hard-coded initialization approach (instead of using $readmem) to load the "demo_program.mem" provided by our TA (Tyler) during the demo session. The memory array is 16 bits per word, and it is addressed using the upper bits of the 16-bit address input since the instructions are 2-byte aligned.

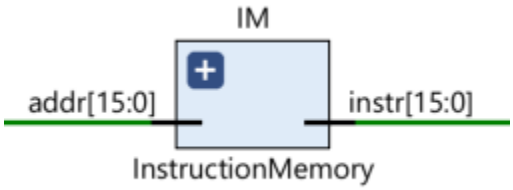


Fig 12. Instruction Memory

- InstructionMemory_TestBench.v:

The InstructionMemory successfully implements the required operations. I just used 8(technically 9) instructions here because we got 28 instructions which are too long. So the simulation results indicate that the module correctly outputs the expected instruction for each valid address.

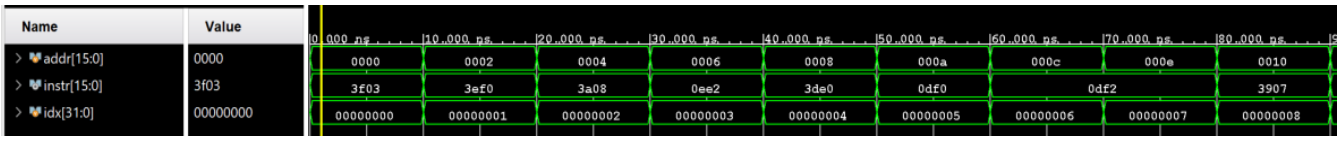


Fig 13. Test Bench for Instruction Memory

Rachel:

- ALU Multiplexer - Mux2.v

This module takes in two 16-bit inputs and uses a control signal that is initialized as **select** to choose what gets passed as input to the ALU, more specifically, it selects what goes into the ALU as the second input. This is essential and useful because it allows the CPU to support two types of instructions, which is time and space efficient. Depending on the binary value on the control signal, it selects either a register value or an immediate value as the second input to the ALU.

- Test and Simulation - aluMux_tb.v

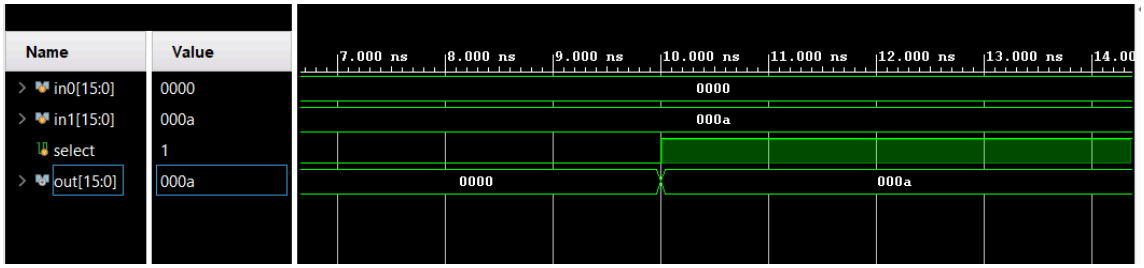


Fig 14. Test Bench for ALU Multiplexer Module

This test has two inputs that could be passed to the ALU. The first input, in0, represents a value stored in a register whereas the second input, in1, is an immediate value. The output is either one of the inputs that gets passed to the ALU.

There are two test cases that verify the behavior of the multiplexer. There is a control signal, select, that chooses which input is passed as the second input to the ALU. If select = 0, then in0 gets passed as second input to the ALU whereas if select = 1, then in1 gets passed as the second input to the ALU.

- DataPath:

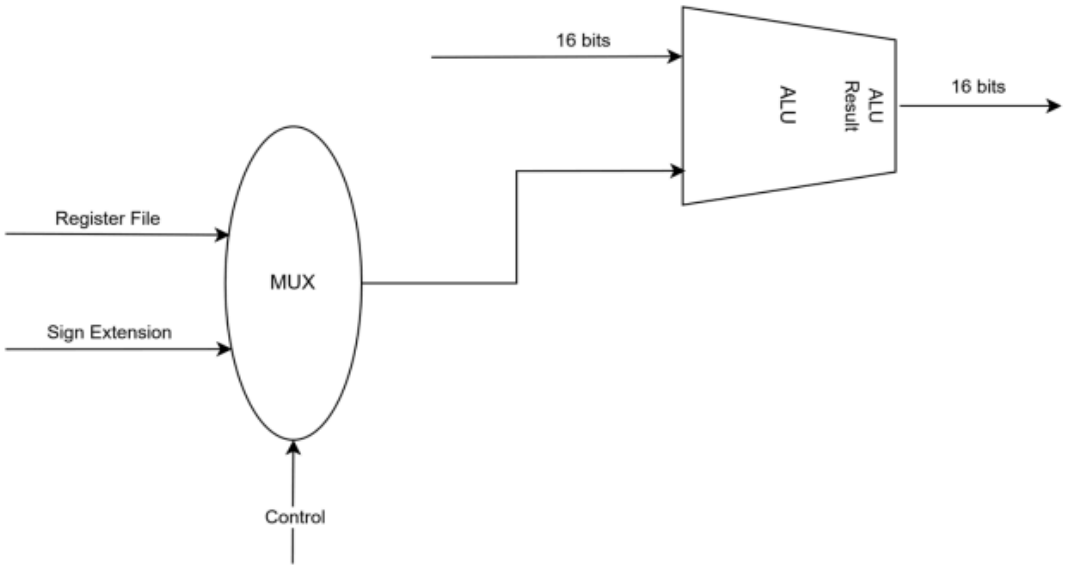


Fig 15. ALU Multiplexer Datapath
(Reference)

- WriteBack Multiplexer - Mux2.v

This module takes in two 16-bit inputs and uses a control signal that is initialized as **select** to choose which input is passed as the input to the register file. It decides what data source gets written back into the register file at the end of an instruction's execution. Depending on the binary value on the control signal, it writes back either the ALU result from arithmetic instructions or data memory from load instructions to the register file. This is important and helpful because it allows support for both memory and ALU instructions and ultimately, it chooses the correct data source to pass to the register file in an efficient way.

- Test and Simulation - wbMux_tb.v

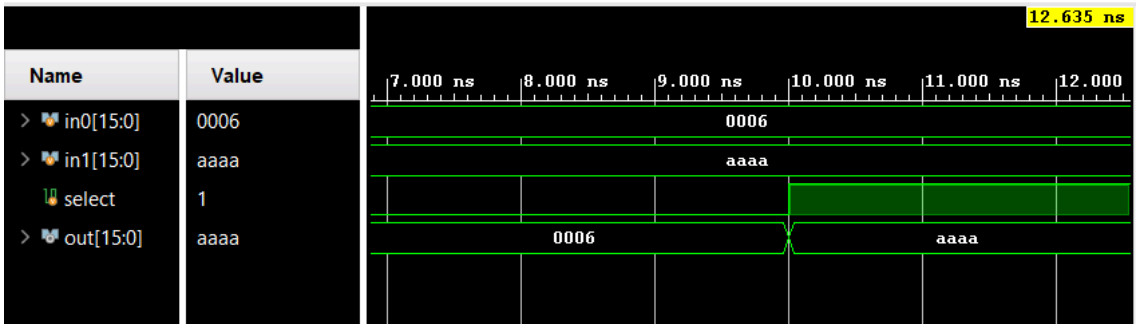


Fig 16. Test Bench for wb Multiplexer

This test has two inputs that could be passed to the register file. The first input, in0, represents an ALU result whereas the second input, in1, is the data memory output. The output is either one of the inputs that gets passed to the register file.
There are two test cases that verify the behavior of the multiplexer. There is a control signal, select, that chooses which input is passed as the second input to the register file. If select = 0, then in0 gets passed as second input to the register file whereas if select = 1, then in1 gets passed as the second input to the register file.

- DataPath:

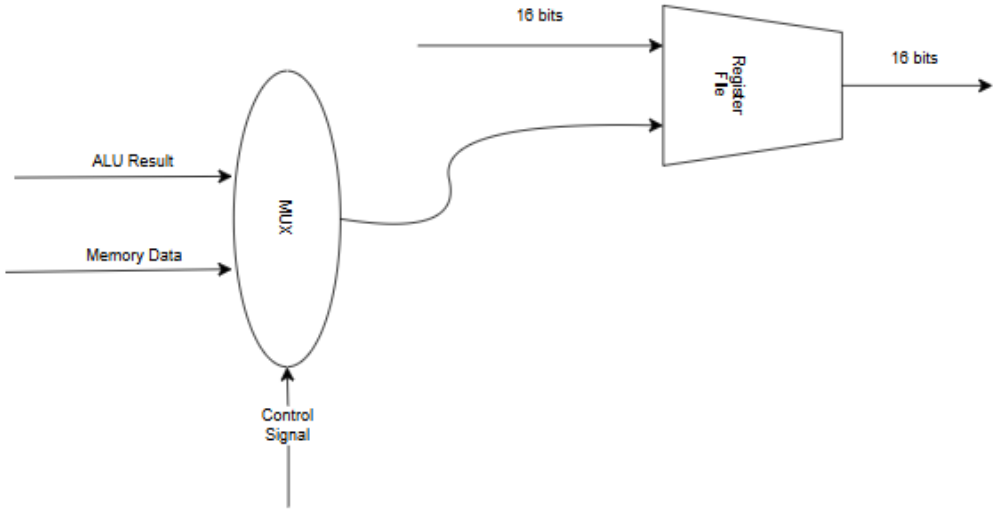


Fig 17. wb Multiplexer Datapath

Xin Virgil Lu:

- RegisterFile.v (Includes 16 register addresses, 16 bits each)

The Register File is a module that manages 16 registers each register is 16 bits wide. In general, the Register File module can read values from two separate register locations and has the ability to write data onto a register location when deemed necessary, specifically when the RegWrite signal is set to 1 as well as the clock signal going from low to high. The module also has the ability to clear all registers to 0 with the reset signal when it is set to 1. This is so that the module can be initialized for another operation. The Register File is solely in charge of reading the data from the two appointed registers and writing the resulting data into the write reg location if needed.

Inputs:

Clock signal
Reset signal
RegWrite signal
Read_reg 1 and 2 [3:0]
Write_reg [3:0] (There are 16 locations and so only 4 bits are needed)
Write_data [15:0] (16 bits of data to write in the resulting address location)

Outputs:

Read_data 1 and 2 [15:0] (16 bits of data per file to read)

Register File Datapath:

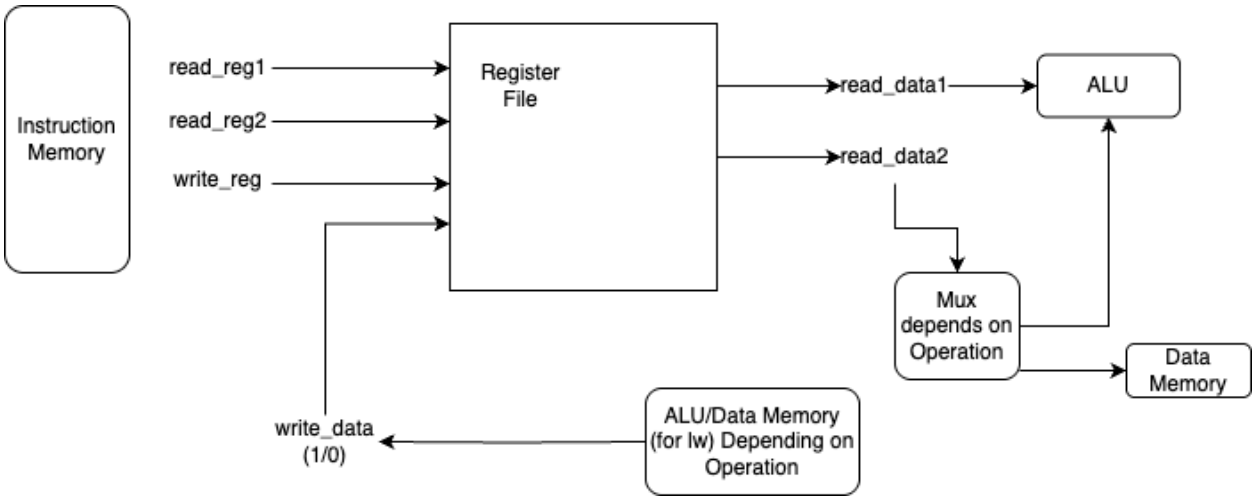


Fig 18. Register File Datapath Reference [2]

Test and Simulation with Register File:

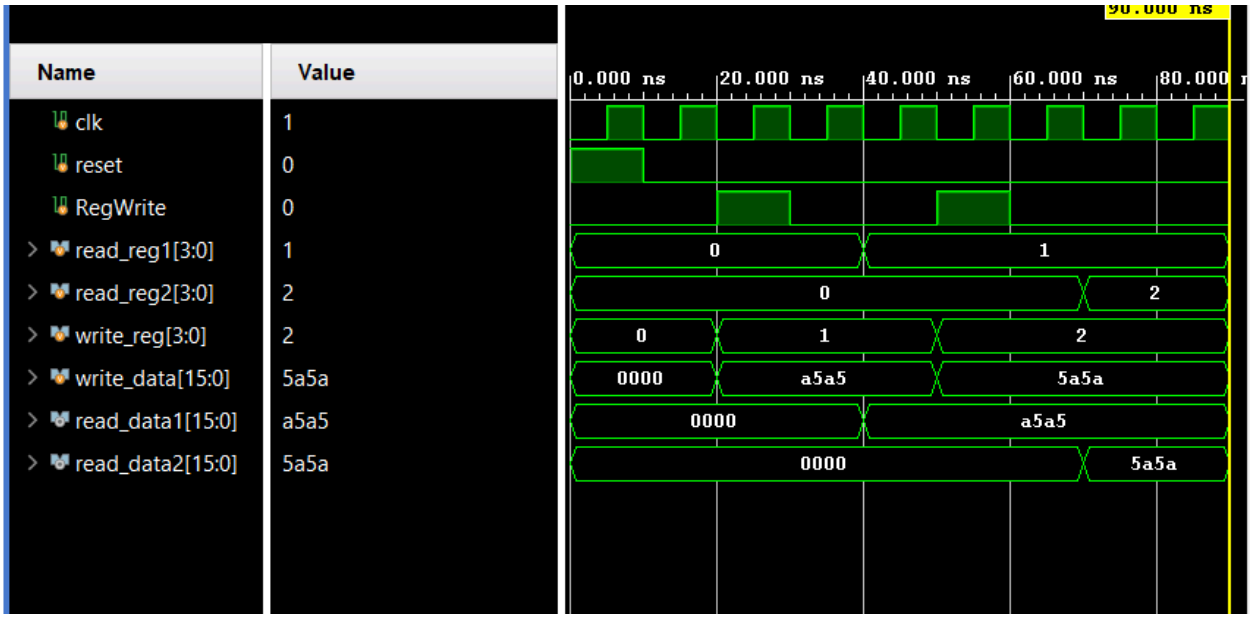


Fig 19. Test bench simulation for File Register Module.

First test was reset, setting everything to 0 therefore initializing the Register File module. Next test, data was written into register location 1 where Register 1 data is

A5A5. read_reg1 was then labelled the location of Register 1 and so read_reg1 has the data of A5A5, Test after was writing data into a different location Register 2 writing in data 5A5A. Afterwards, labelling read_reg2 as the Register 2 location and so read_reg2 data includes 5A5A.

Lucy:

- ProgramCounter.v

The program counter is a register within the processor that holds the memory address of the next instruction to be executed. This allows the CPU to fetch and execute instructions in the same sequence. The CPU uses the PC to determine the correct memory address before an instruction is fetched. Once it is, the PC will increment to point to the next instruction in the program's sequence. In regards to a jump of branch instruction, the PC is updated to point to the new memory address specified by the instruction, allowing the program to deviate from the sequential execution flow.

Inputs:

The inputs of the program counter are clock (clk), reset, PC input, and load. The output is PC output. The clock and reset cycles are inputted to ensure that the PC is only altered or incremented during a clock or reset cycle. The reset cycle in particular resets the PC back to its base value, to allow it to be rerun correctly. PC input is the original memory address inside the PC, while PC output is the new memory address after the PC is changed. Load indicates if a load instruction is being used, meaning the PC will remain unchanged.

- Test and simulation: tb_ProgramCounter.v

The testing began with ensuring that when the reset cycle is executed, the Program Counter resets its value to 0. Afterwards, it goes through a regular clock cycle, which the PC correctly increments. A load instruction is ran through, where the counter remains unchanged, until another clock cycle is run which increments the

PC once more.

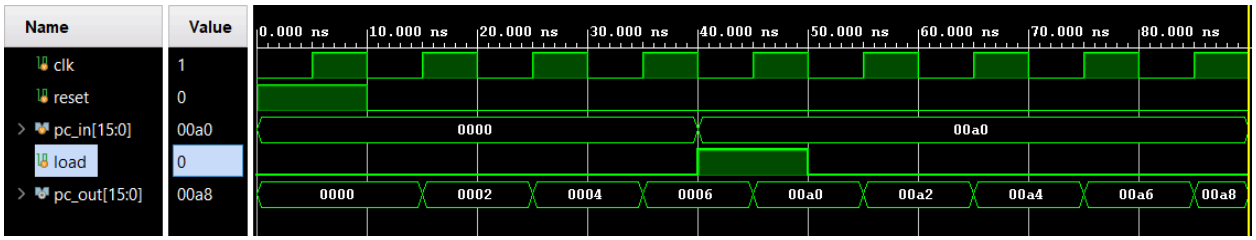


Fig 20. Test bench for Program Counter module

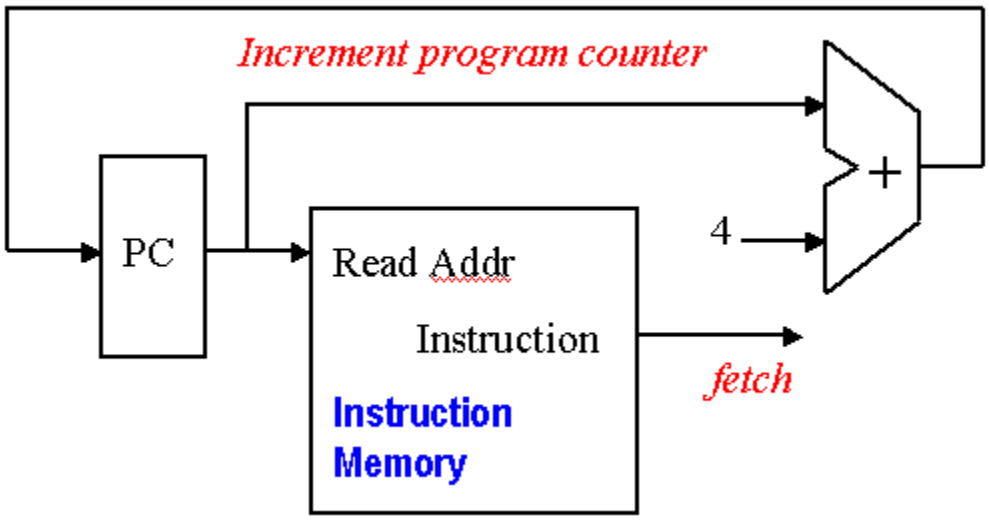


Fig 21. Program Counter Datapath (Reference)

- SignExt4.v

  The sign extension module is the module utilizing sign extension, a process used to increase the number of bits of a signed binary number, typically a two's complement number, while preserving its sign and value by replicating the most significant bit to the left.

  Input:

  The input is a smaller signed number, in this case 8 bits, into a 32-bit signed number.

- Test and simulation: tb_SignExt4.v

  Sign extension was tested on different inputs, shown below:

  in = 4'b0000; #10; // 0x0000
  in = 4'b0111; #10; // 0x0007
  in = 4'b1000; #10; // 0xFFF8

in = 4'b1111; #10; // 0xFFFF
in = 4'b1010; #10; // 0xFFFA

The expected outputs are commented on the right side. After running the test bench, the proper hexadecimal sign-extended values were correctly set to the output.

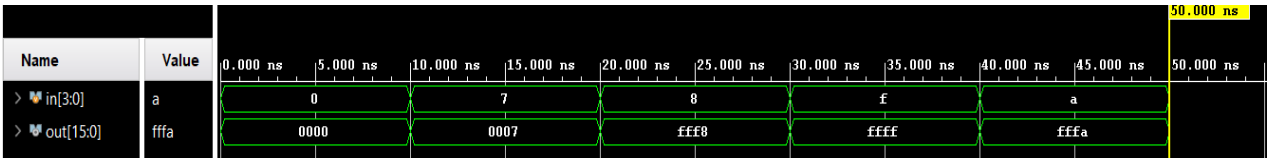


Fig 22. Test bench for Sign Extension module

Sign Extension Datapath:

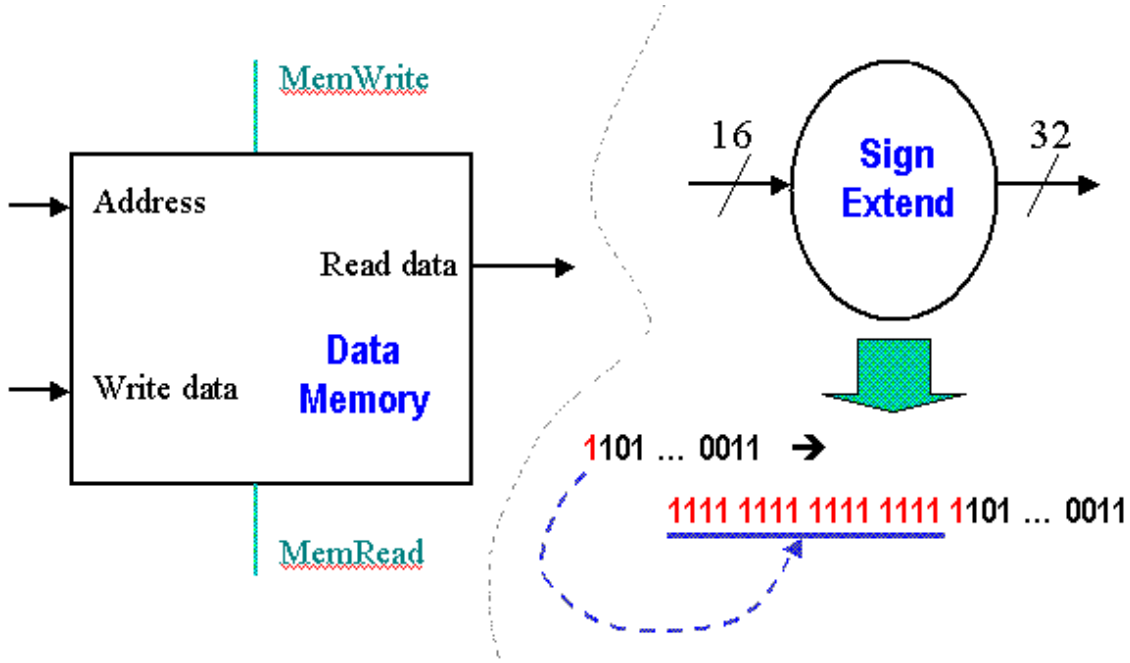


Fig 23. Sign Extension Datapath
(Reference)

5. What We Have Changed

Since our initial submission, we have implemented the following modifications:

- Synchronous Debug Output:
  We modified the Register File and CPU16.v to include a synchronous "debug" register that captures and holds the value that is written to the

destination register. This ensures that the LED output shows the correct value even when the next instruction is being decoded.

- Initialization Enhancements:
  Both the Instruction Memory and Data Memory, as well as the Register File, are now explicitly initialized to 0.
  Detailed Test Bench Documentation:
  We provided individual test benches for each module (ALU, Control Unit, Data Memory, Instruction Memory, Multiplexer) that confirm correct operation.
- Hardware Pin Assignment Update:
  The XDC file has been updated to map the clock, reset, and debug LED signals to the specific pins on the Basys 3 board per the project requirements.

Reference

[1] Computer Organization and Design RIC-V Edition
[2] CSE490_590_Project1_Spring2025_16_bit_processor_pointers
https://cse.buffalo.edu/~rsridhar/cse490-590/hw/CSE490_590_Project1_Spring2025_16_bit_processor_pointers.pdf
[3] https://cse.buffalo.edu/~rsridhar/cse490-590/

Appendix

demo_program.mem:

0011111100000011
0011111011110000
0011101000001000
0000111011100010
0011110111100000
0000110111110000
0000110111110010
0000110111110010
0011100100000111
0011100000000110
0011000100000001
0000100110000011
0000100000010010
0101100100001101
0010111011100000
0010110111100010
0001001011100000
0001001111100001
0001010011100010
0001010111100011
0001011011100100
0100001011100010
0000001011100001
0000010011010001
0101001000000010
0000111011010001
0110111111100110
0110111111111010