Computer Organization Final Project: How do computers actually compute?

1 Group members:

Aneesh Sai Kolukuluri	koluka
Justin Tran	tranj2
Tashi Sherpa	sherpt2
Max Zhang	zhangm21
David Fong	fongd2

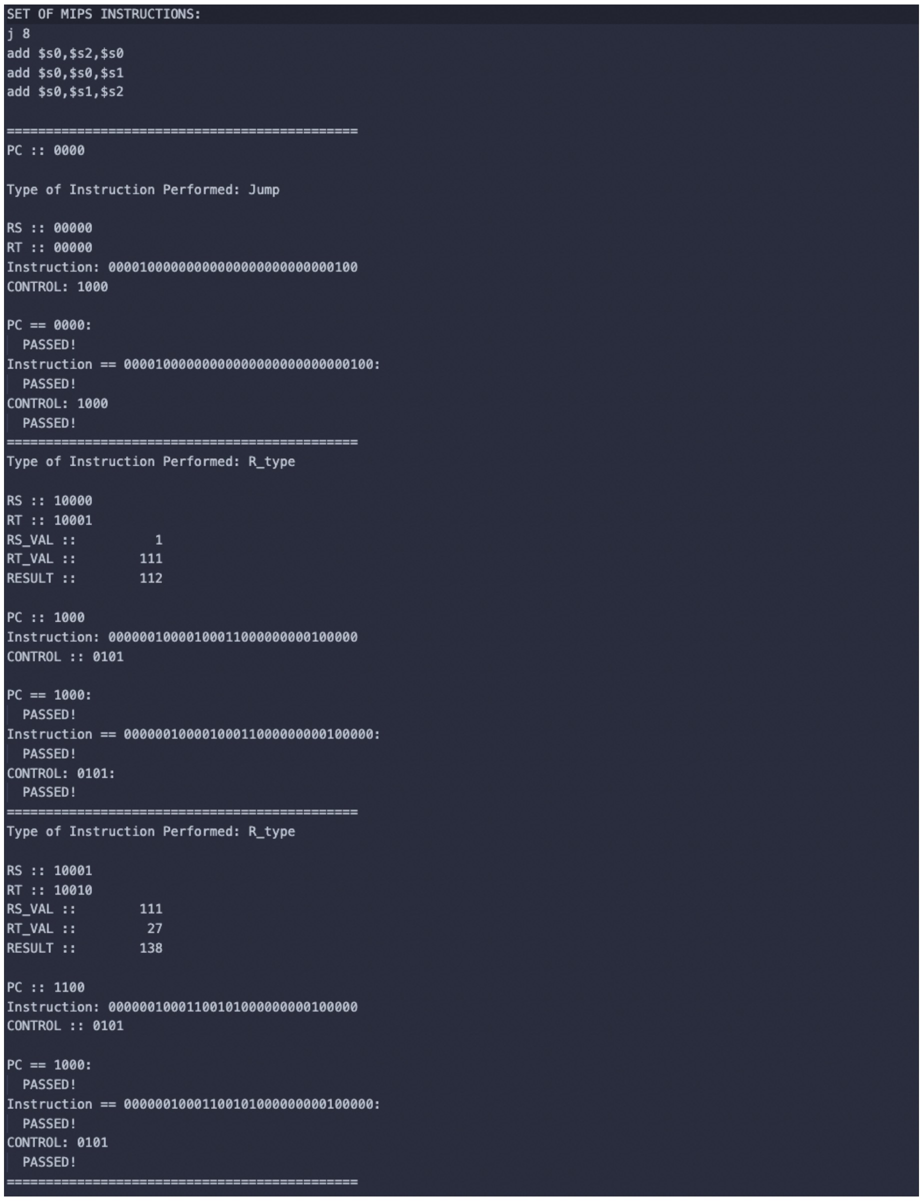
2 Code Outputs:

This code compiles and runs as expected.

We implemented the code from scratch.

Set of MIPs instructions:

```
j 8
add $s0, $s2, $s0
add $s0, $s0, $s1
add $s0, $s1, $s1
```



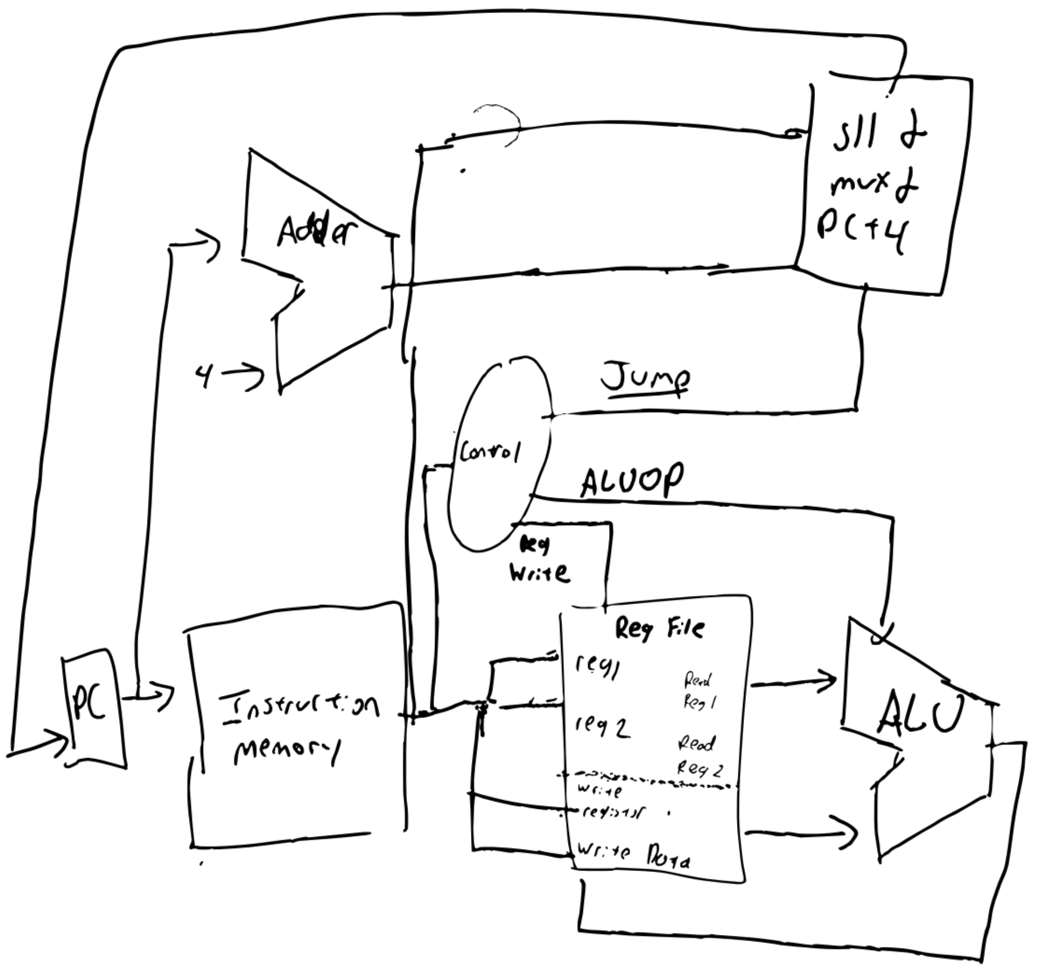
```
SET OF MIPS INSTRUCTIONS:
j 8
add $s0,$s2,$s0
add $s0,$s0,$s1
add $s0,$s1,$s2

=============================================
PC :: 0000

Type of Instruction Performed: Jump

RS :: 00000
RT :: 00000
Instruction: 00001000000000000000000000000100
CONTROL: 1000

PC == 0000:
  PASSED!
Instruction == 00001000000000000000000000000100:
  PASSED!
CONTROL: 1000
  PASSED!
=============================================
Type of Instruction Performed: R_type

RS :: 10000
RT :: 10001
RS_VAL ::          1
RT_VAL ::        111
RESULT ::        112

PC :: 1000
Instruction: 00000010000100011000000000100000
CONTROL :: 0101

PC == 1000:
  PASSED!
Instruction == 00000010000100011000000000100000:
  PASSED!
CONTROL: 0101:
  PASSED!
=============================================
Type of Instruction Performed: R_type

RS :: 10001
RT :: 10010
RS_VAL ::        111
RT_VAL ::         27
RESULT ::        138

PC :: 1100
Instruction: 00000010001100101000000000100000
CONTROL :: 0101

PC == 1000:
  PASSED!
Instruction == 00000010001100101000000000100000:
  PASSED!
CONTROL: 0101
  PASSED!
=============================================
```

3 Design Choices:

We implemented a datapath from scratch with the following characteristics:

- Non-Pipelined
- Single-Level Memory
- Supports R-type (add) and J-Type (jump) instructions.

4 Datapath Description:



- PC
- Adder
- Multiplexor (mux implemented with shift left 2 involved)
- Instruction Memory
- Control
- Register File
- ALU
- Testbench

5 Simulation:

We first define the timescale as 1 ns / 1 ns. This will specify the incrementation of the "clk" variable.

"pc", "v0", "v1", "v2", and "v3" are all "reg" variables, which are crucial because this will ensure they are part of the memory. The variables are saved across instructions, so this is necessary for a single-level memory. We initialize these values in lines 16-21 of our testbench. "pc" is also initialized in lines 28-31. PC, v0, v1, v2, v3 have their own "always" functions in the testbench which are sensitive to the outputs of the datapath, which are "newPC" and "value0"-"value3". "value0"-"value3" represent the data in $s0-$s3 respectively.

Keep in mind that in our testbench:
v0 = 1
v1 = 111
v2 = 27
v3 = 3

For testing purposes, we will be checking values for PC, control and the instruction (declared as wires). For a brief time, we also tested the value that was written back to the register, which is given by "writeData", however we decided that it was more verbose to display the values of the "rs", "rt", and the result in the ALU module when we run our program.

For now, we will explain the process of the first instruction, "j 8".

As explained in the individual component analysis (the sections below), the IM grabs the current PC and the clock counter. When "clk" rises (hence posedge) to 1, the IM will set the corresponding instruction and output it. When clock goes back down to 0 (hence negedge), we will then compute the next PC value in the Adder module. This will be fed to the "PC Mux" module after the "Control" module runs. **This is important so we can ensure that we grab the instruction first BEFORE we increment PC.**

This will be wired to our "Control" module (you can see our wires in the top module!), which has an "always" procedure that is sensitive to "instruction", causing it to run after the instruction fetch.

Control will recognize the instruction as a Jump instruction. This will be displayed in the output. The control should now have bits "1000", and this will be tested in our testbench using our assert function. ALUOp is not used considering that the ALU is not used at all for a jump, so it is set to "00". Jump is obviously "1" and RegWrite is "0". This will ensure that the other modules do not perform any unneeded operations.

The "Pc_mux" module will take in the incremented PC as a wire as well as the instruction. Because Jump = 1, we compute the new PC by adding the distance given by instruction [26:0] which is 4. Doing so will skip the second instruction because PC + 4 + 4 = 8, which skips the instruction corresponding to PC = 4.

Let's now analyze the next instruction, "add $s0, $s0, $s1".

Just like in "Jump", IM reads in the new PC, and Control reads in the instruction, but this time, Control will recognize the add instruction and set the outputs, control, to "0101", ALUOp to "10", Regwrite to "1", and Jump to "0".

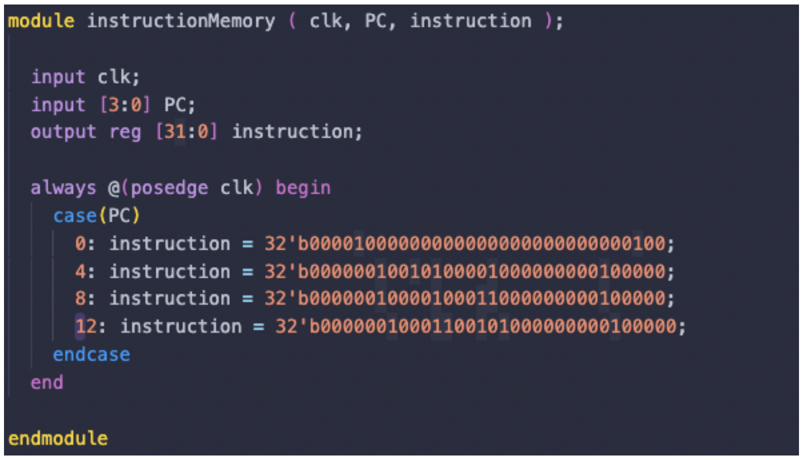
Next, we feed the instruction and initial values (v0-v3) into the "RegFile" module. Instruction comes in the form of rs, rt, and rd, where they are represented by Instruction[25:21] = 10000, Instruction[20:16] = 10000, and Instruction[15:11] = 10001, respectively. Though "rd" is not used, we wanted to declare it a dependency for precautionary measures. As explained in the "Register File" section, we compare "rs" and "rt", and set "read1" and "read2", which are our outputs, to the corresponding values. In our case, rs is "10000", so read1 = v0, and rt is "10001", so read2 = v1.

With both "Control" and "RegFile", because the always function in ALU is sensitive to the values of "rs" and "rt", the ALU module will perform the addition of the values of both registers and feed the sum to "result", which is the wire "ALUresult".

ALUresult is now treated as input for "writeData", which is wired to the "WriteRegFile" module. The always function in the module is sensitive to "writeData," which has not changed until the completion of the ALU, so it will run accordingly in the right order. Our destination register is $s0, which is given by "10000". This means "writeData" will now be stored in "value0" and the rest of the values remain unchanged.

Because Jump is "0", the program counter simply increments by 4 and goes to the next line. Afterwards, the "always" functions defined in the testbench will automatically update the values in v0, v1, v2, and v3 for the next add instruction "add $s0,$s1,$s2" and proceed the same way it did for the previous add instruction with the new updated values.

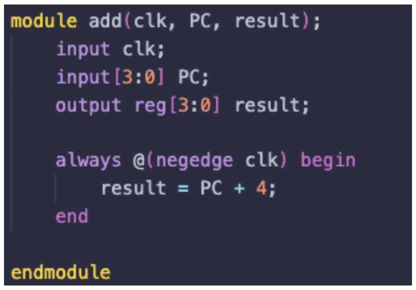
6 Instruction Memory Description



```
module instructionMemory ( clk, PC, instruction );

  input clk;
  input [3:0] PC;
  output reg [31:0] instruction;

  always @(posedge clk) begin
    case(PC)
      0: instruction = 32'b00001000000000000000000000000100;
      4: instruction = 32'b00000010010100001000000000100000;
      8: instruction = 32'b00000010000100011000000000100000;
      12: instruction = 32'b00000010001100101000000000100000;
    endcase
  end

endmodule
```

For instruction memory, we provided the instruction addresses manually and based on the value of the PC, the corresponding instruction is fetched and used throughout the program. The program uses a single-level memory, and thus we do not require the use of data memory because we use the instructions add and jump, which never enter the data memory; For this reason, we didn't implement data memory.

7 Adder

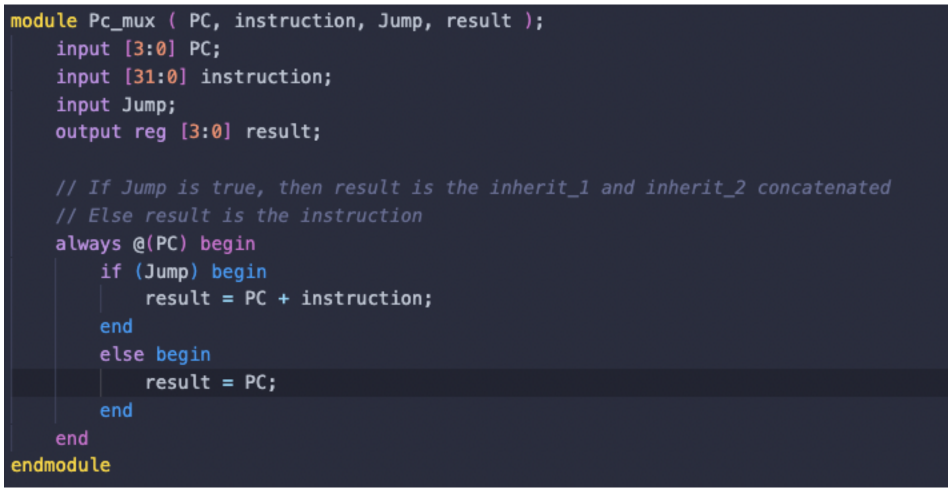


```
module add(clk, PC, result);
    input clk;
    input[3:0] PC;
    output reg[3:0] result;

    always @(negedge clk) begin
        result = PC + 4;
    end

endmodule
```

To increment the PC by 4 for the next instruction, we took in the current PC value as input and have the result stored in the output parameter labeled "result"; this is the equivalent of PC+4. Additionally, the adder takes in the clock as the input, so it updates the result through the always condition; the reason why the condition is "negedge clk" is because whenever we retrieve the new PC–we want the right instruction. Therefore, we want the right instruction first then the PC and since PC updates when clk = 0, in instruction memory–the instructions are updated when clk = 1 or based on "posedge clk".

The computed PC from this module will NOT necessarily be the final output and the next PC for the next instruction. Jump instructions may influence the PC, so the output of this module will be wired to the input of the "PC Mux" for further computation.

8 PC Mux



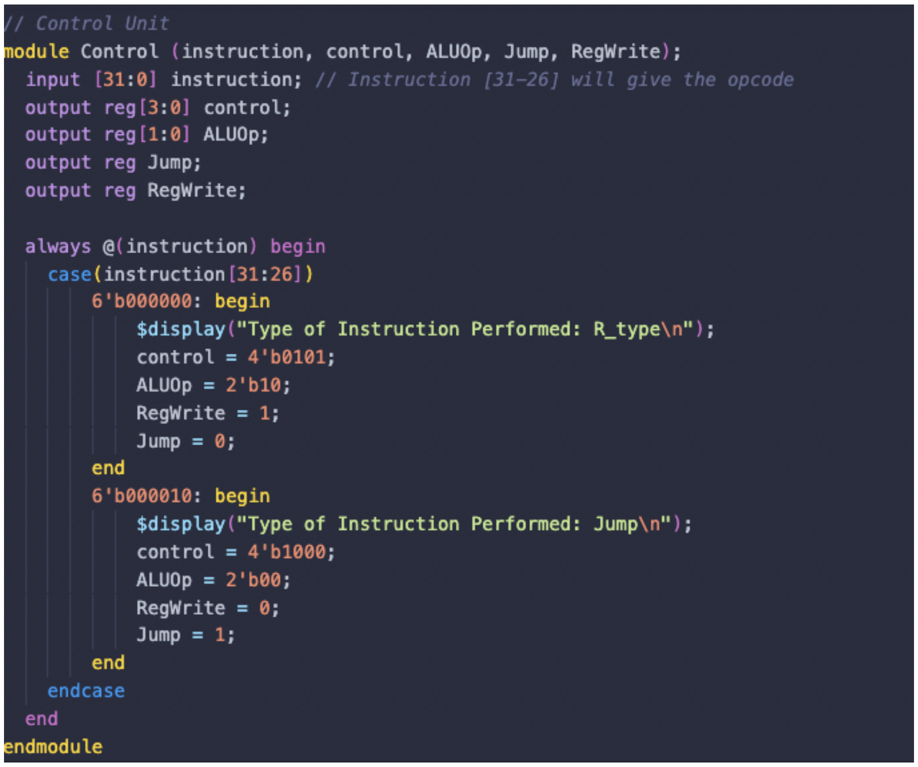
```
module Pc_mux ( PC, instruction, Jump, result );
    input [3:0] PC;
    input [31:0] instruction;
    input Jump;
    output reg [3:0] result;

    // If Jump is true, then result is the inherit_1 and inherit_2 concatenated
    // Else result is the instruction
    always @(PC) begin
        if (Jump) begin
            result = PC + instruction;
        end
        else begin
            result = PC;
        end
    end
endmodule
```

PC Mux takes in PC **(the incremented PC from the Adder module)**, the current instruction, the Jump-bit, and outputs the final PC that will be used for the next MIPS instruction.

If the Jump-Bit is 1, a Jump is occurring. Therefore, we will take the last twenty-six bits of the instruction and calculate the distance from PC+4 to the next PC value. This will ensure that the next PC will jump to the respective instruction.

If the Jump-Bit is 0, then we simply feed in the incremented PC input to the output and proceed as normal, incrementing the PC by 4 to the next MIPS line.

9 Control

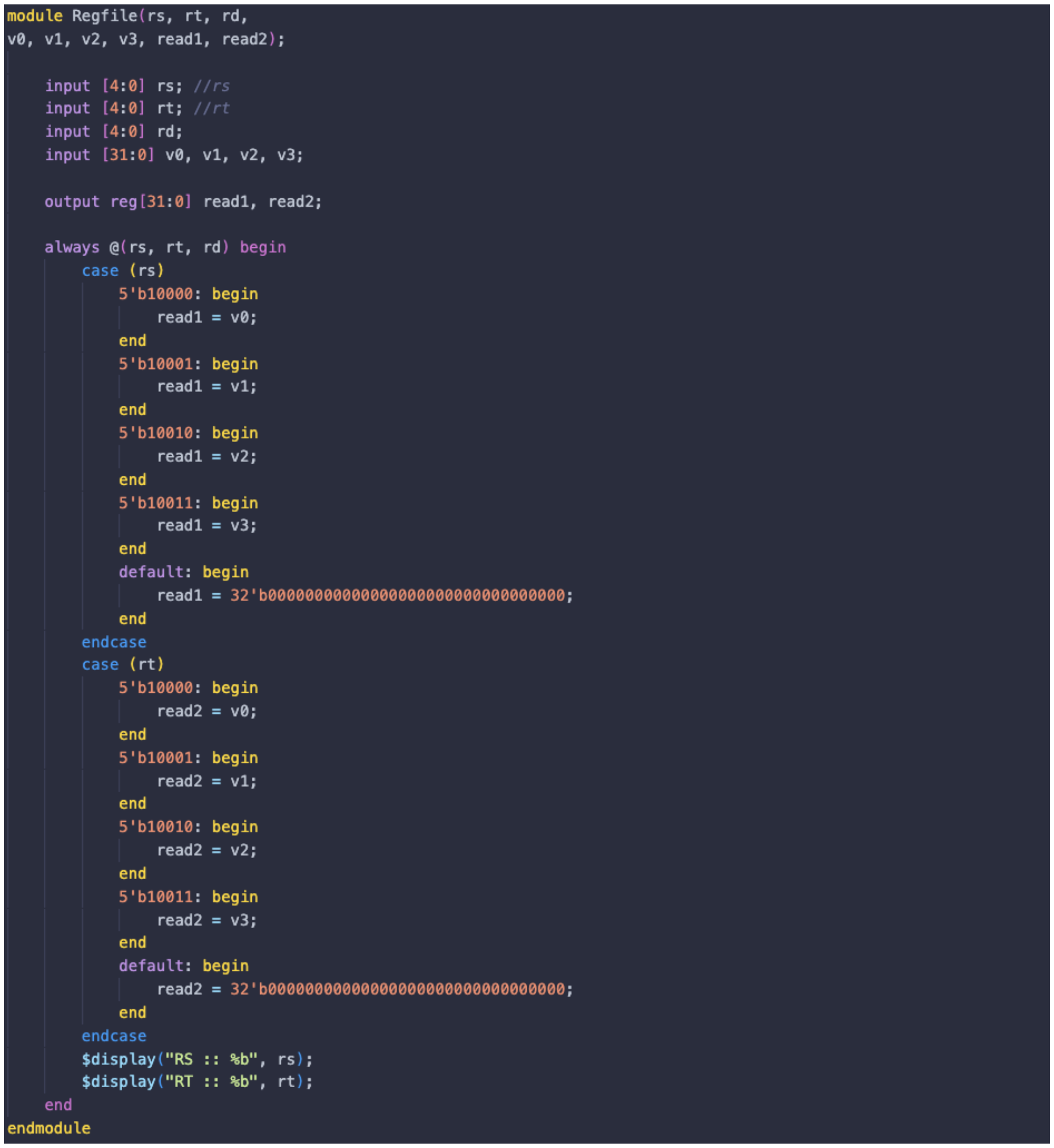


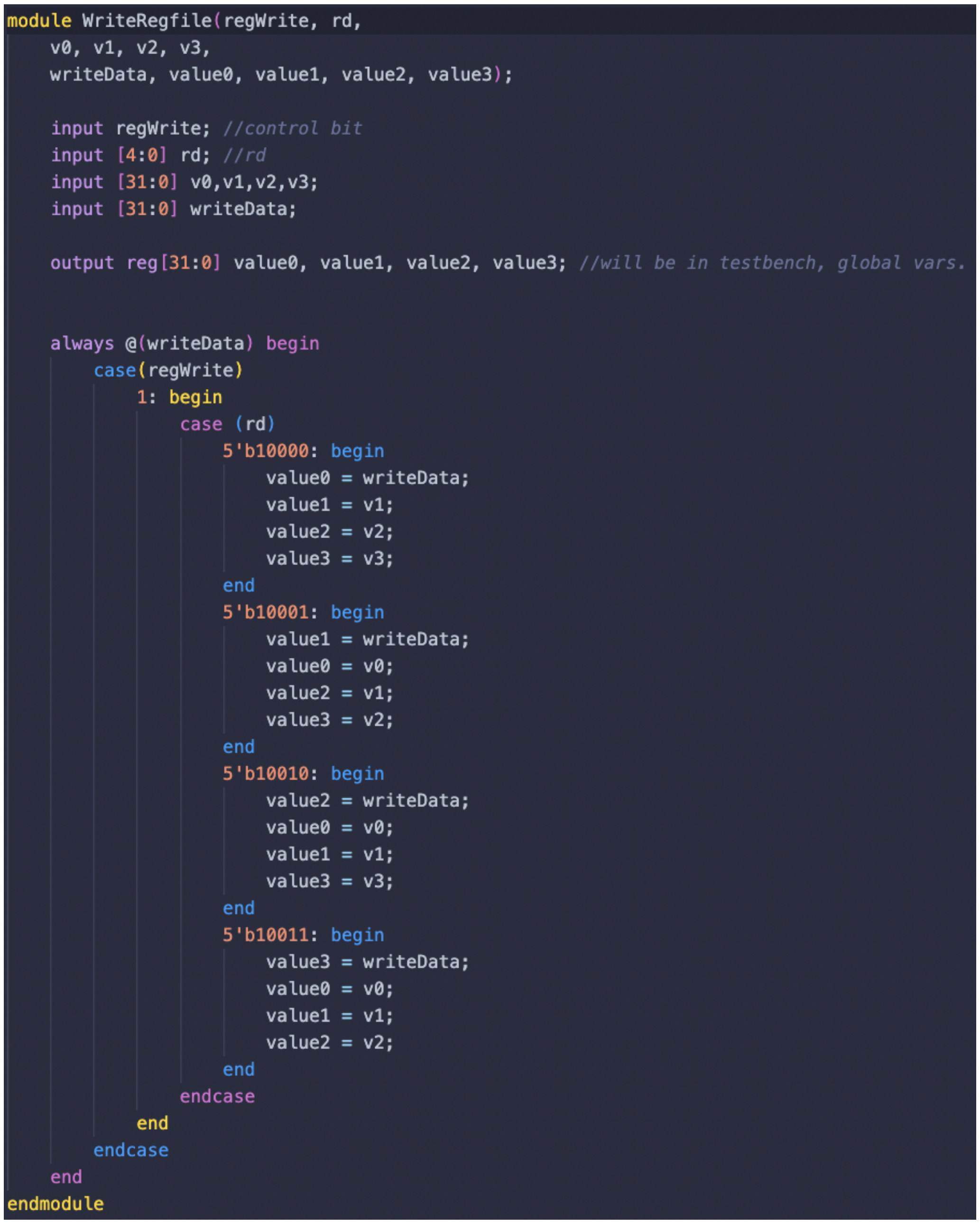
```
// Control Unit
module Control (instruction, control, ALUOp, Jump, RegWrite);
  input [31:0] instruction; // Instruction [31-26] will give the opcode
  output reg[3:0] control;
  output reg[1:0] ALUOp;
  output reg Jump;
  output reg RegWrite;

  always @(instruction) begin
    case(instruction[31:26])
        6'b000000: begin
            $display("Type of Instruction Performed: R_type\n");
            control = 4'b0101;
            ALUOp = 2'b10;
            RegWrite = 1;
            Jump = 0;
        end
        6'b000010: begin
            $display("Type of Instruction Performed: Jump\n");
            control = 4'b1000;
            ALUOp = 2'b00;
            RegWrite = 0;
            Jump = 1;
        end
    endcase
  end
endmodule
```

The control takes in the instruction in order for it to parse the opcode–which is the highest 5 bits to generate the selection bits for the muxs throughout the datapath; thus, respective selection bits are generated for the addition and jump instructions. These are done through case statements where the outputs are the jump signal for the PC mux, the regWrite for the register file to determine if write data is inputted, the ALUop for the ALU to perform addition and nothing for the jump, and all the bits from control for the output.

10 Register File

Note: Our implementation utilizes two functions to separately handle putting data to read data 1 and 2 for ALU computation and writing back (pictures below).


```
module Regfile(rs, rt, rd,
v0, v1, v2, v3, read1, read2);

    input [4:0] rs; //rs
    input [4:0] rt; //rt
    input [4:0] rd;
    input [31:0] v0, v1, v2, v3;

    output reg[31:0] read1, read2;

    always @(rs, rt, rd) begin
        case (rs)
            5'b10000: begin
                read1 = v0;
            end
            5'b10001: begin
                read1 = v1;
            end
            5'b10010: begin
                read1 = v2;
            end
            5'b10011: begin
                read1 = v3;
            end
            default: begin
                read1 = 32'b00000000000000000000000000000000;
            end
        endcase
        case (rt)
            5'b10000: begin
                read2 = v0;
            end
            5'b10001: begin
                read2 = v1;
            end
            5'b10010: begin
                read2 = v2;
            end
            5'b10011: begin
                read2 = v3;
            end
            default: begin
                read2 = 32'b00000000000000000000000000000000;
            end
        endcase
        $display("RS :: %b", rs);
        $display("RT :: %b", rt);
    end
endmodule
```



```
module WriteRegfile(regWrite, rd,
    v0, v1, v2, v3,
    writeData, value0, value1, value2, value3);

    input regWrite; //control bit
    input [4:0] rd; //rd
    input [31:0] v0,v1,v2,v3;
    input [31:0] writeData;

    output reg[31:0] value0, value1, value2, value3; //will be in testbench, global vars.


    always @(writeData) begin
        case(regWrite)
            1: begin
                case (rd)
                    5'b10000: begin
                        value0 = writeData;
                        value1 = v1;
                        value2 = v2;
                        value3 = v3;
                    end
                    5'b10001: begin
                        value1 = writeData;
                        value0 = v0;
                        value2 = v1;
                        value3 = v2;
                    end
                    5'b10010: begin
                        value2 = writeData;
                        value0 = v0;
                        value1 = v1;
                        value3 = v3;
                    end
                    5'b10011: begin
                        value3 = writeData;
                        value0 = v0;
                        value1 = v1;
                        value2 = v2;
                    end
                endcase
            end
        endcase
    end
endmodule
```

Our register file unit effectively acts as our datapath facilitator. It feeds in rs and rt into read data 1 and 2 and handles our single level memory (by storing the ALU result into a global register and keeping our two copies of global registers equal).

For register file unit, we receive the control bits (more specifically, we are looking for regWrite) and three sections of the instruction (rs, rt, and rd). We are also taking in rd because our implementation only supports R and J type instructions, so there aren't instructions such as lw and sw that store the result into rt.
Then we take the values of rs and rt and compare them to store them accordingly into our global input variables, v0,v1,v2,v3 via two case statements. We also have a default statement to store the value as 0 (for testing purposes).

After, we assign the values rs and rt into read data 1 and read data2, which are both output regs, in preparation of ALU.
After we've calculated the result of ALU, we take in the output of ALU, which is writeData (a wire), and write it into one of the registers.

We've also made a few changes to accurately represent a non-pipelined, single memory datapath while maintaining its integrity.

Changes to Register File Unit

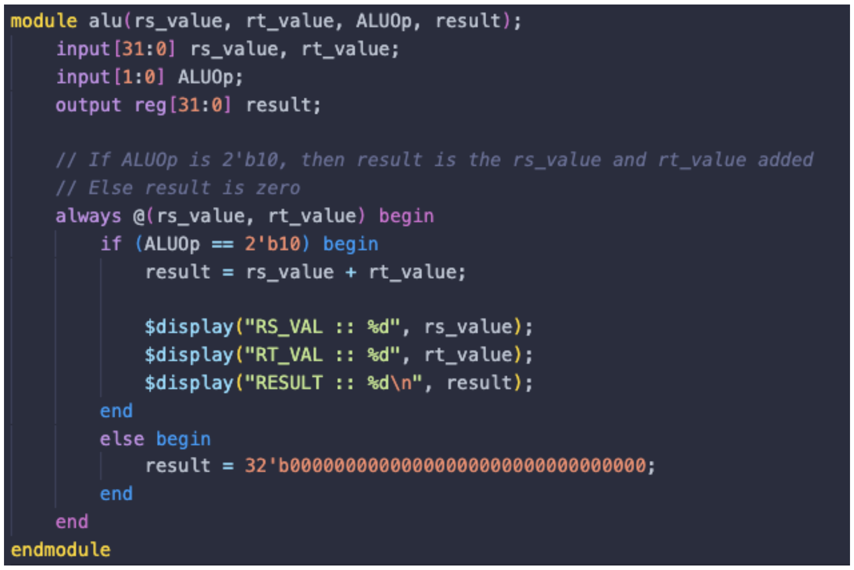
1. Turning the register file unit into two: one read unit and one write back unit.
2. Utilizing two forms of single memory (effectively the same); one is used for the ID stage reading input from rs and rt and the other for storing write back data input.

For our first change, we separated it to two components, a write register and a read register.
The reason why we did this is because while implementing the singular register unit, we ran across a problem: the register unit was expected to handle both read and write under two separate always statements (one handling the rs, rt, rd elements upon input, and the other handling the ALU output as input).
*This is the equivalent of a structural hazard; we had no way of effectively separating both the reading and writing (ID and WB) stages.

For our second change, we utilized two forms of data memory: v0,v1,v2,v3 (which is input memory that gets initialized with test values at the beginning of each cycle) and value0,value1,value2,value3 (which is output memory that changes with the write back value).
The reason why we did this is because a value in verilog cannot act as both an input and output (meaning it cannot be effectively a left hand and right hand side value), so we had to improvise to make register file a middle man.

In our testbench, we have an always block that continously sets the values of value0, value1, value2, and value3 (which are updated at the end of each instruction) equal to v0, v1, v2, and v3.

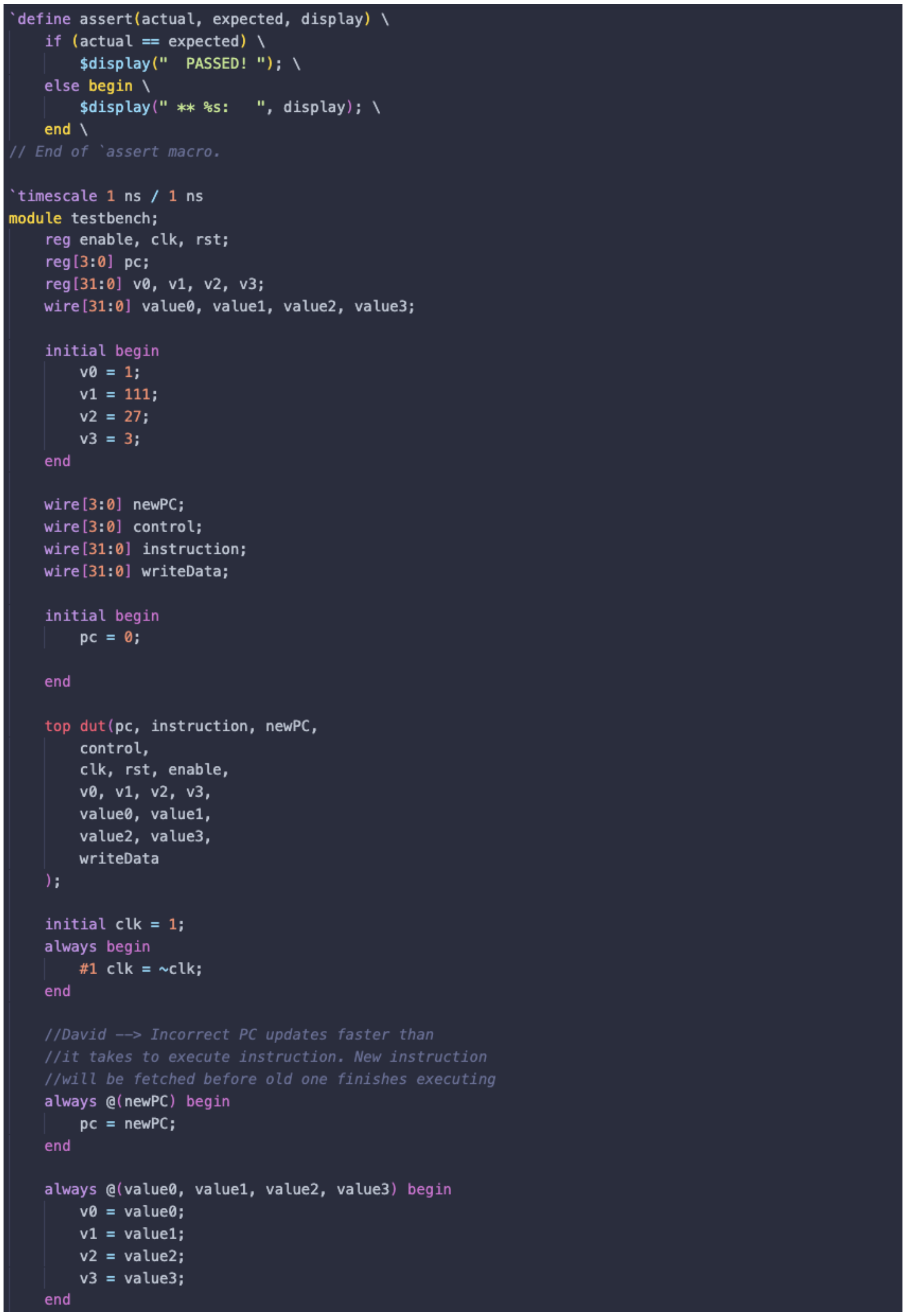
11 ALU



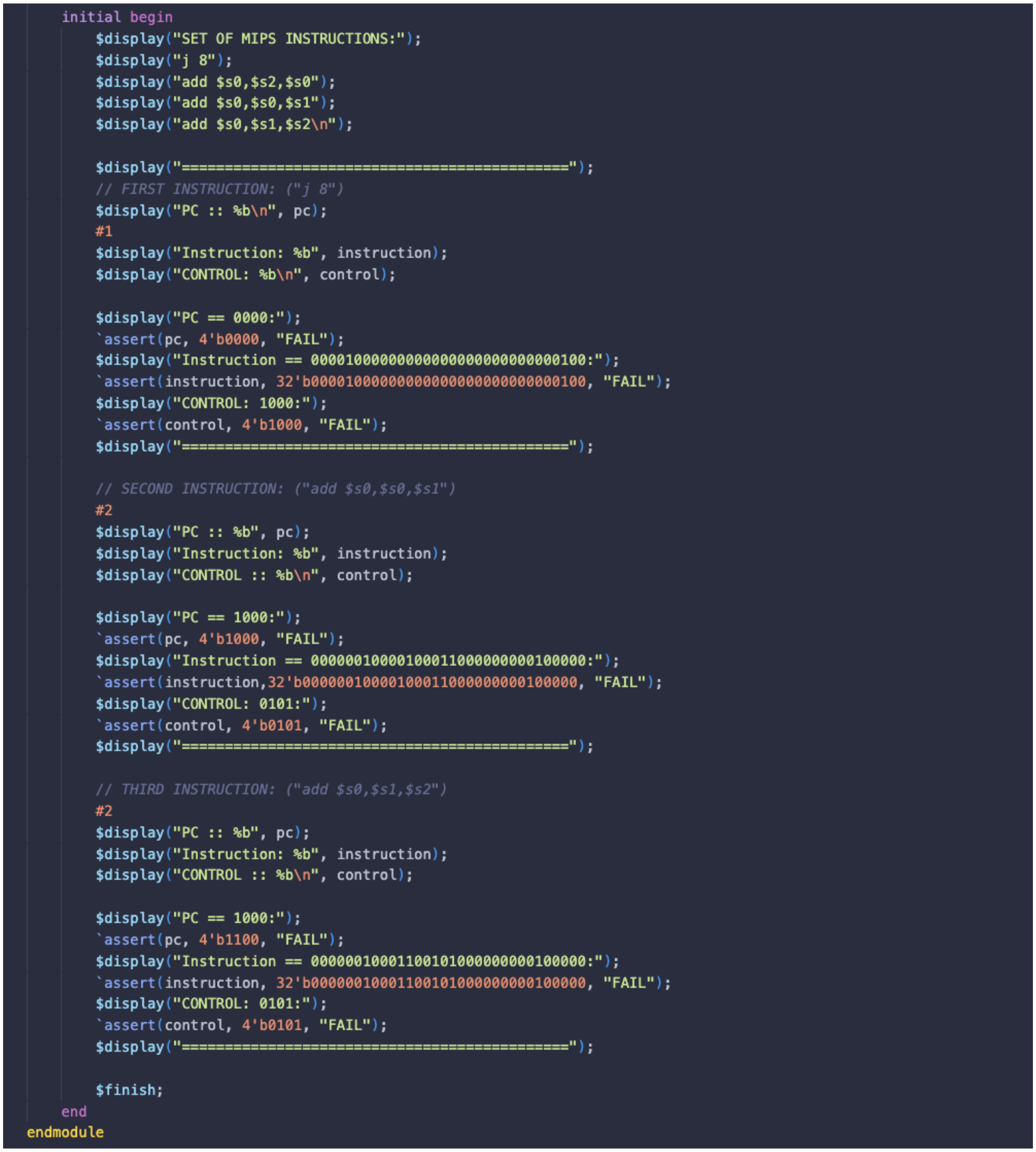
```verilog
module alu(rs_value, rt_value, ALUOp, result);
    input[31:0] rs_value, rt_value;
    input[1:0] ALUOp;
    output reg[31:0] result;

    // If ALUOp is 2'b10, then result is the rs_value and rt_value added
    // Else result is zero
    always @(rs_value, rt_value) begin
        if (ALUOp == 2'b10) begin
            result = rs_value + rt_value;

            $display("RS_VAL :: %d", rs_value);
            $display("RT_VAL :: %d", rt_value);
            $display("RESULT :: %d\n", result);
        end
        else begin
            result = 32'b00000000000000000000000000000000;
        end
    end
endmodule
```

Since the ALU only needs to perform addition for the values of the registers, it takes in the rs–the first register and the rt–the second register which is stored in the result–the rd as output. Additionally, we take in the ALUop as input to distinguish the ALU for when addition is performed versus jump–which makes the result the address. Specifically, for the implementation, ALU is performed whenever the rs and rt values are changed; hence, the always keyword and we check through conditionals for the operations.

12 Testbench



```
`define assert(actual, expected, display) \
    if (actual == expected) \
        $display("  PASSED! "); \
    else begin \
        $display(" ** %s:   ", display); \
    end \
// End of `assert macro.

`timescale 1 ns / 1 ns
module testbench;
    reg enable, clk, rst;
    reg[3:0] pc;
    reg[31:0] v0, v1, v2, v3;
    wire[31:0] value0, value1, value2, value3;

    initial begin
        v0 = 1;
        v1 = 111;
        v2 = 27;
        v3 = 3;
    end

    wire[3:0] newPC;
    wire[3:0] control;
    wire[31:0] instruction;
    wire[31:0] writeData;

    initial begin
        pc = 0;

    end

    top dut(pc, instruction, newPC,
        control,
        clk, rst, enable,
        v0, v1, v2, v3,
        value0, value1,
        value2, value3,
        writeData
    );

    initial clk = 1;
    always begin
        #1 clk = ~clk;
    end

    //David --> Incorrect PC updates faster than
    //it takes to execute instruction. New instruction
    //will be fetched before old one finishes executing
    always @(newPC) begin
        pc = newPC;
    end

    always @(value0, value1, value2, value3) begin
        v0 = value0;
        v1 = value1;
        v2 = value2;
        v3 = value3;
    end
```



```
    initial begin
        $display("SET OF MIPS INSTRUCTIONS:");
        $display("j 8");
        $display("add $s0,$s2,$s0");
        $display("add $s0,$s0,$s1");
        $display("add $s0,$s1,$s2\n");

        $display("=============================================");
        // FIRST INSTRUCTION: ("j 8")
        $display("PC :: %b\n", pc);
        #1
        $display("Instruction: %b", instruction);
        $display("CONTROL: %b\n", control);

        $display("PC == 0000:");
        `assert(pc, 4'b0000, "FAIL");
        $display("Instruction == 00001000000000000000000000000100:");
        `assert(instruction, 32'b00001000000000000000000000000100, "FAIL");
        $display("CONTROL: 1000:");
        `assert(control, 4'b1000, "FAIL");
        $display("=============================================");

        // SECOND INSTRUCTION: ("add $s0,$s0,$s1")
        #2
        $display("PC :: %b", pc);
        $display("Instruction: %b", instruction);
        $display("CONTROL :: %b\n", control);

        $display("PC == 1000:");
        `assert(pc, 4'b1000, "FAIL");
        $display("Instruction == 00000010000100011000000000100000:");
        `assert(instruction,32'b00000010000100011000000000100000, "FAIL");
        $display("CONTROL: 0101:");
        `assert(control, 4'b0101, "FAIL");
        $display("=============================================");

        // THIRD INSTRUCTION: ("add $s0,$s1,$s2")
        #2
        $display("PC :: %b", pc);
        $display("Instruction: %b", instruction);
        $display("CONTROL :: %b\n", control);

        $display("PC == 1000:");
        `assert(pc, 4'b1100, "FAIL");
        $display("Instruction == 00000010001100101000000000100000:");
        `assert(instruction, 32'b00000010001100101000000000100000, "FAIL");
        $display("CONTROL: 0101:");
        `assert(control, 4'b0101, "FAIL");
        $display("=============================================");

        $finish;
    end
endmodule
```

13 IDE, VsCode, and Iverilog

To run the code we all downloaded iVerilog (Icarus verilog).
We run the code from the visual studio code terminal.

This was possible by the command (sudo apt install iverilog) and the VSCode extension:

```
Verilog-HDL/SystemVerilog/Bluespec SystemVerilog
```

The command to run the code is:

```
iverilog -o test *.v
vvp test
```

We coded in verilog using vscode as our editor.
* The output is provided in a .txt file called "output.txt" if needed.

14 Contributions:

Aneesh: I worked mainly on the Register file modules, top module, PC update, and Test-bench. I also worked on the timing, and output display. I did all this with David and Tashi. Justin and Max worked on other parts in the beginning, but eventually, we all started working together. I worked a little on everything and also did the write-up with my teammates.

Justin: I came up with the implementation of the ALU and Adder from scratch and contributed to the instruction memory and to a portion of the testbench and the top module. Along the way I encountered issues regarding the ordering of the code and certain syntactical errors, but those were resolved by my peers–Max and Aneesh.

Max:
Together with Justin, I also mainly worked on ALU and adder. Initially, we had a lot of errors, and I helped resolve that by suggesting different program structure such as splitting the Register module into 2 components and also fixing major syntax errors. I also helped design the testbench by implementing the "clk" variable and ensured that all the modules ran in the correct order.

Tashi: I worked on the register file unit and write register file unit with Aneesh. Particularly, we had to re-engineer this unit from a simple register file unit (in a normal datapath) to two units: one for reading and one for writing. I helped pioneer this implementation. I also worked on the main Top module and helped connect wires and outputs, as well as the timing for the test bench.

David: I worked with Tashi, Justin, Max, and Aneesh. I worked mainly on the Register File, and Instruction Memory. I also helped with planning on the logic of each of the components by drawing out the modified data path diagram. I also contributed to debugging as well as writing the Latex File.

15 References

- Zybooks code for lab 10 problem 11.15.1(specifically the use of top and testbench1.v)
- Zybooks week 11 full chapter (used as study material)
- https://hdlbits.01xz.net/wiki/Main_Page
- IDE: VSCode
- Icarus Verilog - http://iverilog.icarus.com/

16 Communication

We had a GitHub repository to manage our code.
We had a discord to communicate meeting times, share resources, and add updates to the code.